Språkbehandling och datalingvistik

Projektarbeten 2003

Handledare: Pierre Nugues och Richard Johansson

LUNDS UNIVERSITET

Institutionen för Datavetenskap

http://www.cs.lth.se

Printed in Sweden
Lund 2004

Innehåll

LUNDS UNIVERSITET

Institutionen för Datavetenskap

http://www.cs.lth.se

GroupDetector

Jimmy Andersson
Lunds universitet
jimmy77@telia.com

Tommy Karlsson
Lunds universitet
tommy.karlsson.350@student.lu.se

1 Inledning

Som projektuppgift valde vi att göra ett program för att leta upp verb, substantiv, adjektiv och prepositionsgrupper i texter.
Mjukvaran som utvecklats i vårt projekt ska användas av CarSim-projektet som är ett system för att automatiskt konvertera skrivna beskrivningar av trafikolyckor till 3D-animeringar. CarSim projektet utvecklas av LUCAS (Center for Applied Software Research vid Lunds Universitet).
Andra användningsområden är grammatikkontroll, översättarstöd och informationsextrahering.

2 Taggar

2.1 Granska-taggar

I en text taggad enligt granska-formatet åtföljs varje ord av en tagg som beskriver vilken satsdel ordet tillhör. Exempel 2.1 visar en mening som är taggad enligt granska-formatet. För en förklaring av taggarnas betydelse, se bilaga 1.

```
Hans <ps.utr/neu.sin/plu.def> tidigare <jj.kom.utr/neu.sin/plu.ind/def.nom> grafik
<nn.utr.sin.ind.nom> har <vb.prs.akt> varit <vb.sup.akt> noggrant <ab.pos>
genomtänkt <pc.prf.utr.sin.ind.nom> och <kn> konstruerad <pc.prf.utr.sin.ind.nom>
lika <ab> målmedvetet <ab.pos> som <kn> den <dt.utr.sin.def> arkitektur
<nn.utr.sin.ind.nom> den <pn.utr.sin.def.sub/obj> skildrar <vb.prs.akt> .
```

Exempel 2.1

2.2 SUC1A-taggar

SUC1A-formatet har en något annorlunda utseende, jämför med granska-formatet. För varje ord finns information om satsdel, ordets grundform och ordningsnummer. Exempel 2.2 visar en mening som är taggad enligt SUC1A-formatet. För en förklaring av taggarnas betydelse, se bilaga 1.

```
("<Någonting>"      <140>
        (PN NEU SIN IND SUB/OBJ "någonting"))
("<har>"      <141>
        (VB PRS AKT "ha"))
("<hänt>"     <142>
        (VB SUP AKT "hända"))
("<med>"      <143>
        (PP "med"))
```

```
("<hans>"       <144>
          (PS UTR/NEU SIN/PLU DEF "hans"))
("<syn>"        <145>
          (NN UTR SIN IND NOM "syn"))
("<på>"         <146>
          (PP "på"))
("<staden>"     <147>
          (NN UTR SIN DEF NOM "stad"))
("<som>"        <148>
          (HP - - - "som"))
("<också>"      <149>
          (AB "också"))
("<fått>"       <150>
          (VB SUP AKT "få"))
("<följder>"    <151>
          (NN UTR PLU IND NOM "följd"))
("<för>"        <152>
          (PP "för"))
("<hans>"       <153>
          (PS UTR/NEU SIN/PLU DEF "hans"))
("<grafik>"     <154>
          (NN UTR SIN IND NOM "grafik"))
("<.>"  <155>
          (DL MAD "."))
```
Exempel 2.2

2.3 Våra taggar
Våra taggar beskriver olika grupper av ord i texten. Början på en grupp markeras med en tagg på formen <XX> och avslutas med </XX>. Exempel 2.3 visar en mening som är taggad enligt vårt format.

Taggarna är följande:
- AG: adjektivgrupp
- PG: prepositionsgrupp
- NG: substantivgrupp
- VG: verbgrupp

```
<AG> Omkullvält </AG> <PG> i Östeuropa </PG> och illa ute <PG> i
Sovjetunionen </PG> <VG> dyker </VG> <NG> den f_d kuppmakaren </NG> och
<NG> frälsargestalten </NG> upp <PG> i 117 olika skepnader </PG> <VG> där
</VG> <NG> granskogen </NG> <VG> susar </VG> och <NG> sjön </NG> <VG>
ligger </VG> <AG> blank </AG> inte långt <PG> från Jönköping </PG> .
```
Exempel 2.3

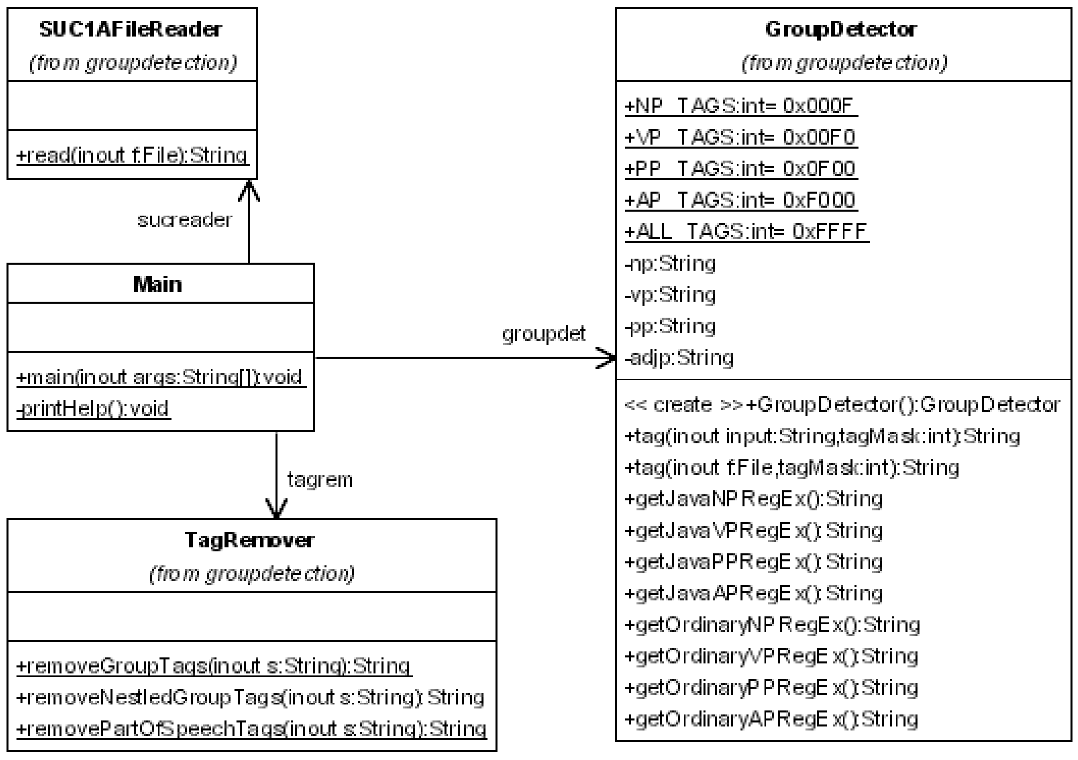
3 Systemet

3.1 Struktur

Programmet är uppdelat i 3 olika klasser, *GroupDetector*, *TagRemover* och *SUC1AFileReader*.

- Klassen GroupDetector används för att leta upp ordgrupper i en text.
- Klassen TagRemover används för att ta bort taggar i en text så att texten blir lättare att läsa.
- Klassen SUC1AFileReader används för att läsa in filer med SUC1A-formatet och formatera om det till granska-formatet.

Main-klassen används för att testa de olika systemen och de olika klasserna.


SUC1AFileReader
(from groupdetection)
+read(inout f:File):String
sucreader
Main
+main(inout args:String[]):void
-printHelp():void
groupdet
tagrem
TagRemover
(from groupdetection)
+removeGroupTags(inout s:String):String
+removeNestledGroupTags(inout s:String):String
+removePartOfSpeechTags(inout s:String):String
GroupDetector
(from groupdetection)
+NP_TAGS:int= 0x000F
+VP_TAGS:int= 0x00F0
+PP_TAGS:int= 0x0F00
+AP_TAGS:int= 0xF000
+ALL_TAGS:int= 0xFFFF
-np:String
-vp:String
-pp:String
-adjp:String
<< create >>+GroupDetector():GroupDetector
+tag(inout input:String,tagMask:int):String
+tag(inout f:File,tagMask:int):String
+getJavaNPRegEx():String
+getJavaVPRegEx():String
+getJavaPPRegEx():String
+getJavaAPRegEx():String
+getOrdinaryNPRegEx():String
+getOrdinaryVPRegEx():String
+getOrdinaryPPRegEx():String
+getOrdinaryAPRegEx():String

Figur 3.1 – Klassdiagram

3.2 Reguljära uttryck

- NP = [[([DET]+) [PRON]+] | [DET ADJ] | [([DET]+) (ADVPOS) (NUM) (ADJP) (NUM) [NOUN]+] | [(DET) POSS_PRON (ADJP) [NOUN]+] ;
- VP = [[INT_REL_ADV] | [(INF) [VERB]+ (PART)] | [AUX NP [VERB]+ (PART)]] ;
- PP = [[[PREP] | [PREP KONJ PREP]] NP] ;
- AP = [([ADV]+) [ADJ]+] ;

4 Utvärdering
För att utvärdera projektet har vi beräknat värden för precision och recall enligt följande formler;

precision = antal korrekta / antal försök
recall = antal korrekta / totalt antal

Vi fick följande resultat:
Precision: 0.949152542372881
Recall: 0.96551724137931

Texten som vi använde vid mätningen finns i bilaga 3.
Vi är väldigt nöjda med resultatet. Det blev lite bättre än vi hade förväntat oss men man ska nog testa med större texter innan man kan dra alltför stora slutsatser.

5 Referenser
Steven Abney, *Chunk Stylebook*, 1996
http://www.vinartus.net/spa/96i.pdf

Victoria Johansson, *NP-detektion*, 2000
http://www.nada.kth.se/theory/projects/granska/rapporter/vicuppsats.pdf

Beáta Megyesi & Sara Rydin, *Towards a Finite-State Parser for Swedish*,
http://www.speech.kth.se/%7Ebea/final-megyesi-rydin.pdf

6 Bilaga 1 - Taggar i Granska & SUC

Category Code	Category
AB	Adverb
DL	Delimiter (Punctuation)
DT	Determiner
HA	Interrogative/Relative Adverb
HD	Interrogative/Relative Determiner
HP	Interrogative/Relative Pronoun
HS	Interrogative/Relative Possessive
IE	Infinitive Marker
IN	Interjection
JJ	Adjective
KN	Conjunction
NN	Noun
PC	Participle
PL	Particle
PM	Proper Noun
PN	Pronoun
PP	Preposition
PS	Possessive
RG	Cardinal Number
RO	Ordinal Number
SN	Subjunction
UO	Foreign Word
VB	Verb

Feature Code	Feature	
UTR	Common (Utrum)	Gender
NEU	Neutre	Gender
MAS	Masculine	Gender
UTR/NEU	Underspecified	Gender
-	Unspecified	Gender
SIN	Singular	Number
PLU	Plural	Number
SIN/PLU	Underspecified	Number
-	Unspecified	Number
IND	Indefinite	Definiteness
DEF	Definite	Definiteness
IND/DEF	Underspecified	Definiteness
-	Unspecified	Definiteness
NOM	Nominative	Case
GEN	Genitive	Case
SMS	Compound	Case
-	Unspecified	Case
POS	Positive	Degree
KOM	Comparative	Degree
SUV	Superlative	Degree
SUB	Subject	Pronoun Form
OBJ	Object	Pronoun Form
SUB/OBJ	Underspecified	Pronoun Form
PRS	Present	Verb Form
PRT	Preterite	Verb Form
INF	Infinitive	Verb Form
SUP	Supinum	Verb Form
IMP	Imperative	Verb Form
AKT	Active	Voice
SFO	S-form	Voice
KON	Subjunctive	Mood
PRF	Perfect	Perfect
AN	Abbreviation	Form

7 Bilaga 2 – Reguljära uttryck

NP

```
(((([\S]+) <dt[^>]*>\s+)?((([\S]+) <pn[^>]*>\s+)+))|((([\S]+) <dt[^>]*>\s+)(([\S]+)
<((jj)|(pc\.prf))[^>]*>\s+))|((([\S]+) <dt[^>]*>\s+)?(([\S]+)
<ab.pos[^>]*>\s+)?((([\S]+) <r[^>]*>\s+)+)?(((([\S]+) <ab[^>]*>\s+)+)?((([\S]+)
<((jj)|(pc\.prf))[^>]*>\s+)+))?((([\S]+) <r[^>]*>\s+)+)?((([\S]+)
<((nn)|(pm))[^>]*>\s+)+))|((([\S]+) <dt[^>]*>\s+)?(([\S]+) <ps[^>]*>\s+)(((([\S]+)
<ab[^>]*>\s+)+)?((([\S]+) <((jj)|(pc\.prf))[^>]*>\s+)+))?((([\S]+)
<((nn)|(pm))[^>]*>\s+)+)))+
```

VP

```
((([\S]+) <ha[^>]*>\s+))|((([\S]+) <ie[^>]*>\s+)?((([\S]+) <vb[^>]*>\s+)+)(([\S]+)
<pc[^>]*>\s+)?)|((([\S]+) <vb[^>]*>\s+)(((([\S]+) <dt[^>]*>\s+)?((([\S]+)
<pn[^>]*>\s+)+))|((([\S]+) <dt[^>]*>\s+)(([\S]+) <((jj)|(pc\.prf))[^>]*>\s+))|((([\S]+)
<dt[^>]*>\s+)?(([\S]+) <ab.pos[^>]*>\s+)?((([\S]+) <r[^>]*>\s+)+)?(((([\S]+)
<ab[^>]*>\s+)+)?((([\S]+) <((jj)|(pc\.prf))[^>]*>\s+)+))?((([\S]+)
<r[^>]*>\s+)+)?((([\S]+) <((nn)|(pm))[^>]*>\s+)+))|((([\S]+) <dt[^>]*>\s+)?(([\S]+)
<ps[^>]*>\s+)(((([\S]+) <ab[^>]*>\s+)+)?((([\S]+)
<((jj)|(pc\.prf))[^>]*>\s+)+))?((([\S]+) <((nn)|(pm))[^>]*>\s+)+)))+((([\S]+)
<vb[^>]*>\s+)+)(([\S]+) <pc[^>]*>\s+)?)
```

AP

```
(((([\S]+) <ab[^>]*>\s+)+)?((([\S]+) <((jj)|(pc\.prf))[^>]*>\s+)+))
```

PP

```
(((([\S]+) <pp[^>]*>\s+))|((([\S]+) <pp[^>]*>\s+)(([\S]+) <kn[^>]*>\s+)(([\S]+)
<pp[^>]*>\s+)))(((([\S]+) <dt[^>]*>\s+)?((([\S]+) <pn[^>]*>\s+)+))|((([\S]+)
<dt[^>]*>\s+)(([\S]+) <((jj)|(pc\.prf))[^>]*>\s+))|((([\S]+) <dt[^>]*>\s+)?(([\S]+)
<ab.pos[^>]*>\s+)?((([\S]+) <r[^>]*>\s+)+)?(((([\S]+) <ab[^>]*>\s+)+)?((([\S]+)
<((jj)|(pc\.prf))[^>]*>\s+)+))?((([\S]+) <r[^>]*>\s+)+)?((([\S]+)
<((nn)|(pm))[^>]*>\s+)+))|((([\S]+) <dt[^>]*>\s+)?(([\S]+) <ps[^>]*>\s+)(((([\S]+)
<ab[^>]*>\s+)+)?((([\S]+) <((jj)|(pc\.prf))[^>]*>\s+)+))?((([\S]+)
<((nn)|(pm))[^>]*>\s+)+)))+
```

8 Bilaga 3 – Test-text
Skillnaden mellan vårt resultat och facit har markerats med fet stil.

Facit
<NG> Syfte </NG> och <NG> fr}gest{llningar </NG> . Mer {n <NG> ett kvarts sekel </NG> <VG> har passerat </VG> sedan <NG> de f|rsta jugoslaverna </NG> <VG> slog </VG> <NG> sig </NG> ner <PG> i Stockholm </PG> (1) . D} , <PG> vid mitten </PG> <PG> av 60-talet </PG> , <VG> s}gs </VG> <NG> arbetskraftsinvandringen </NG> mest som <NG> en tillf{llig l|sning </NG> <PG> p} tillf{lliga problem </PG> ; efter <VG> att ha tj{nat </VG> ihop tillr{ckligt <VG> skulle </VG> <NG> de flesta </NG> <VG> flytta </VG> hem igen . S} <VG> var </VG> <NG> det </NG> <VG> t{nkt </VG> , b}de **<PG> fr}n svenskt </PG>** och <NG> jugoslaviskt myndighetsh}ll </NG> . Men <NG> verkligheten </NG> <VG> blev </VG> **<AG> en annan </AG>** . Bara <NG> n}gra f} </NG> <VG> v{nde </VG> hem igen . [ven om <NG> m}nga </NG> {nnu <VG> har </VG> <NG> starka band </NG> <PG> till hemlandet </PG> och <NG> n}gra </NG> {nnu <VG> n{r </VG> <NG> dr|mmen </NG> <PG> om }terv{ndandet </PG> , s} <VG> har </VG> <NG> de flesta </NG> {nd} <VG> valt </VG> <VG> att bos{tta </VG> <NG> sig </NG> <PG> i Stockholm </PG> **f|r gott** . <PG> Vid slutet </PG> <PG> av 1980-talet </PG> <VG> bodde </VG> <NG> drygt 8000 jugoslaviska medborgare </NG> <PG> i Stockholmstrakten </PG> . <NG> En stor del serber </NG> , men d{rut|ver ocks} <NG> kroater </NG> , <NG> bosnier </NG> , <NG> slovener </NG> , <NG> makedonier </NG> , <NG> montenegriner </NG> , <NG> ungrare </NG> , <NG> vlacher </NG> , <NG> albaner </NG> , <NG> rusiner </NG> och <NG> slovaker </NG>

Vårt resultat
<NG> Syfte </NG> och <NG> fr}gest{llningar </NG> . Mer {n <NG> ett kvarts sekel </NG> <VG> har passerat </VG> sedan <NG> de f|rsta jugoslaverna </NG> <VG> slog </VG> <NG> sig </NG> ner <PG> i Stockholm </PG> (1) . D} , <PG> vid mitten </PG> <PG> av 60-talet </PG> , <VG> s}gs </VG> <NG> arbetskraftsinvandringen </NG> mest som <NG> en tillf{llig l|sning </NG> <PG> p} tillf{lliga problem </PG> ; efter <VG> att ha tj{nat </VG> ihop tillr{ckligt <VG> skulle </VG> <NG> de flesta </NG> <VG> flytta </VG> hem igen . S} <VG> var </VG> <NG> det </NG> <VG> t{nkt </VG> , b}de **fr}n <AG> svenskt </AG>** och <NG> jugoslaviskt myndighetsh}ll </NG> . Men <NG> verkligheten </NG> <VG> blev </VG> **<NG> en annan </NG>** . Bara <NG> n}gra f} </NG> <VG> v{nde </VG> hem igen . [ven om <NG> m}nga </NG> {nnu <VG> har </VG> <NG> starka band </NG> <PG> till hemlandet </PG> och <NG> n}gra </NG> {nnu <VG> n{r </VG> <NG> dr|mmen </NG> <PG> om }terv{ndandet </PG> , s} <VG> har </VG> <NG> de flesta </NG> {nd} <VG> valt </VG> <VG> att bos{tta </VG> <NG> sig </NG> <PG> i Stockholm </PG> **f|r <AG> gott </AG>** . <PG> Vid slutet </PG> <PG> av 1980-talet </PG> <VG> bodde </VG> <NG> drygt 8000 jugoslaviska medborgare </NG> <PG> i Stockholmstrakten </PG> . <NG> En stor del serber </NG> , men d{rut|ver ocks} <NG> kroater </NG> , <NG> bosnier </NG> , <NG> slovener </NG> , <NG> makedonier </NG> , <NG> montenegriner </NG> , <NG> ungrare </NG> , <NG> vlacher </NG> , <NG> albaner </NG> , <NG> rusiner </NG> och <NG> slovaker </NG>

9 Bilaga 4 - Källkod

Här följer källkoden i projektet.

GroupDetector

```
package groupdetection;
import java.io.*;
import java.util.regex.*;
import java.util.*;

/**
 * <p>Title: Noun and Verb group detector</p>
 * <p>Description: This class supplies methods for adding NG and VG tags to
a part-of-speech tagged text.</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: Lund University</p>
 * @author Tommy Karlsson & Jimmy Andersson
 * @version 1.0
 */

public class GroupDetector {
  public static final int NP_TAGS  = 0x000F;
  public static final int VP_TAGS  = 0x00F0;
  public static final int PP_TAGS  = 0x0F00;
  public static final int AP_TAGS  = 0xF000;
  public static final int ALL_TAGS = 0xFFFF;

  private String np;   // beskriver det regexp som används för att matcha
noun-phrase
  private String vp;   // beskriver det regexp som används för att matcha
verb-phrase
  private String pp;   // beskriver det regexp som används för att matcha
preposition-phrase
  private String adjp; // beskriver det regexp som används för att matcha
adjective-phrase


  /**
   * Sets up the strings used to compile the regular expressions.
   */
  public GroupDetector() {

    // matchar ett ord
    String anyword="(?:[\\S]+)";
    // matchar ett adverb
    String adverb="(?:"+anyword+" <ab[^>]*>\\s+)";
    // matchar ab.pos
    String adverb_pos="(?:"+anyword+" <ab.pos[^>]*>\\s+)";
    // matchar ett interrogativit / relativt adverb
    String inter_rel_adverb="(?:"+anyword+" <ha[^>]*>\\s+)";
    // matchar ett adjektiv
    String adjective="(?:"+anyword+" <(?:(?:jj)|(?:pc\\.prf))[^>]*>\\s+)";
    // matchar en preposition
    String prep="(?:"+anyword+" <pp[^>]*>\\s+)";
    // matchar en konjunktion
    String konj="(?:"+anyword+" <kn[^>]*>\\s+)";
    // matchar en sekvens av pronomen
    String pronouns="(?:(?:"+anyword+" <pn[^>]*>\\s+)+)";
    // matchar ett possesivt pronomen
    String posspron="(?:"+anyword+" <ps[^>]*>\\s+)";
    // matchar en determinant
    String det="(?:"+anyword+" <dt[^>]*>\\s+)";
    // matchar en sekvens av räkneord
    String numerals="(?:(?:"+anyword+" <r[^>]*>\\s+)+)";
    // matchar en sekvens av substantiv
    String nouns="(?:(?:"+anyword+" <(?:(?:nn)|(?:pm))[^>]*>\\s+)+)";
    // matchar en infinitiv-markör
    String inf="(?:"+anyword+" <ie[^>]*>\\s+)";
    // matchar en sekvens av verb
    String verbs="(?:(?:"+anyword+" <vb[^>]*>\\s+)+)";
    // matchar participle
    String participle="(?:"+anyword+" <pc[^>]*>\\s+)";
    // matchar aux
    String aux="(?:"+anyword+" <vb[^>]*>\\s+)";
    // matchar en adj-phrase
    adjp="(?:(?:"+adverb+"+)?(?:"+adjective+"+))";
    // matchar en noun-phrase
    np =
"(?:(?:"+det+"?"+pronouns+")|(?:"+det+adjective+")|(?:"+det+"?"+adverb_pos+"
?"+numerals+"?"+adjp+"?"+numerals+"?"+nouns+")|(?:"+det+"?"+posspron+adjp+"?
"+nouns+"))+";
    // matchar en verb-phrase

vp="(?:"+inter_rel_adverb+")|(?:"+inf+"?"+verbs+participle+"?)|(?:"+aux+np+v
erbs+participle+"?)";
    // matchar en prep-phrase
    pp="(?:(?:"+prep+")|(?:"+prep+konj+prep+"))"+np;

//    define NP [ [([DET]+) [PRON]+] | [DET ADJ] | [([DET]+) (ADVPOS) (NUM)
(ADJP) (NUM) [NOUN]+] | [(DET) POSS_PRON (ADJP) [NOUN]+] ;
//    define VP [ [INT_REL_ADV] | [(INF) [VERB]+ (PART)] | [AUX NP [VERB]+
(PART)]] ;
//    define PP [ [[PREP] | [PREP KONJ PREP]] NP] ;
//    define AP [ ([ADV]+) [ADJ]+] ;
  }

  /**
   * Detects noun- and verb-groups in a text, and add appropriate tags.
   * All original tags and text are left untouched.
   *
   * @param s The text to be tagged.
   * @param tagMask The mask of the tags to be added.
   * @return The text with the specified tags.
   */
  public String tag(String input, int tagMask) {
    String taggedString="";
    Vector tag_vector=new Vector();

    try {
      Pattern p;
      Matcher m;
      if((tagMask & NP_TAGS)==NP_TAGS) {
        // kompilera np-regex och skapa en matcher
        p = Pattern.compile(np);
        m = p.matcher(input);
        // applicera np-regex på strängen och spara alla taggar på np-
stacken
        while (m.find()) {
          tag_vector.add(new GroupTag(GroupTag.NG_BEGIN, m.start()));
          tag_vector.add(new GroupTag(GroupTag.NG_END, m.end()));
        }
      }
      if((tagMask & VP_TAGS)==VP_TAGS) {
        // kompilera vp-regex och skapa en matcher
        p = Pattern.compile(vp);
        m = p.matcher(input);
        // applicera vp-regex på strängen och spara alla taggar på vp-
stacken
        while (m.find()) {
          tag_vector.add(new GroupTag(GroupTag.VG_BEGIN, m.start()));
          tag_vector.add(new GroupTag(GroupTag.VG_END, m.end()));
        }
      }
      if((tagMask & PP_TAGS)==PP_TAGS) {
        // kompilera pp-regex och skapa en matcher
        p = Pattern.compile(pp);
        m = p.matcher(input);
        // applicera pp-regex på strängen och spara alla taggar på pp-
stacken
        while (m.find()) {
          tag_vector.add(new GroupTag(GroupTag.PP_BEGIN, m.start()));
          tag_vector.add(new GroupTag(GroupTag.PP_END, m.end()));
        }
      }
      if((tagMask & AP_TAGS)==AP_TAGS) {
        // kompilera adjp-regex och skapa en matcher
        p = Pattern.compile(adjp);
        m = p.matcher(input);
        // applicera adjp-regex på strängen och spara alla taggar på pp-
stacken
        while (m.find()) {
          tag_vector.add(new GroupTag(GroupTag.ADJP_BEGIN, m.start()));
          tag_vector.add(new GroupTag(GroupTag.ADJP_END, m.end()));
        }
      }

      // joxa lite för att sortera taggarna
      tag_vector.trimToSize();
      Object[] tagv = tag_vector.toArray();
      Arrays.sort(tagv);
      List tag_list=Arrays.asList(tagv);

      // lägg till taggarna i strängen, börja bakifrån och arbeta mot
strängens början.
      GroupTag g;
      for(int i=tag_list.size()-1;i>=0;i--) {
        g=(GroupTag)tag_list.get(i);
        input=input.substring(0,g.getPos()) + g +
input.substring(g.getPos());
      }
      taggedString = input;
    }
    catch(Exception e) {System.out.println(e.getMessage());
e.printStackTrace();}
    return taggedString;
  }

  /**
   * Adds support for tagging text stored in a file.
   * The idea is obviously to capture the file-reading inside this class.
   * Any user of this method is responsible for supplying a file with
correctly formatted text.
   * Note that the actual tagging is performed by a call to {@link
GroupDetector.tag(String) tag(String)}, hence
   * this method should deliver exactly the same result as if you do the
file-reading yourself.
   * Also note that this method has a limitation on file-size: files may not
be any larger than 2^31-1 bytes.
   *
   * @param f The file with the text to be tagged.
   * @param tagMask The mask of the tags to be added.
   * @return The text with the specified tags.
   *
   */
  public String tag(File f, int tagMask) {
    // check if file is readable
    if(!f.canRead()) {
      System.out.println("Can't read file: "+f.getAbsolutePath());
      return "";
    }
    try {
      // read the file
      FileReader fr=new FileReader(f);
      char[] cv=new char[(int)f.length()];
      fr.read(cv);
      fr.close();
      String s=new String(cv);
      // do the tagging
      return tag(s, tagMask);
    }
    catch(Exception e) { System.out.println(e.getMessage()); return "";}
  }

  /**
   * Get the java-formatted string describing the regexp used to match noun-
phrases.
   * @return The actual string used to compile the regexp.
   */
  public String getJavaNPRegEx() {
    return np;
  }

  /**
   * Get the java-formatted string describing the regexp used to match verb-
phrases.
   * @return The actual string used to compile the regexp.
   */
  public String getJavaVPRegEx() {
    return vp;
  }

  /**
   * Get the java-formatted string describing the regexp used to match
pronoun-phrases.
   * @return The actual string used to compile the regexp.
   */
  public String getJavaPPRegEx() {
    return pp;
  }

  /**
   * Get the java-formatted string describing the regexp used to match
adjective-phrases.
   * @return The actual string used to compile the regexp.
   */
  public String getJavaAPRegEx() {
    return adjp;
  }

  /**
   * Get a perl-like version of the regexp used to match noun-phrases.
   * Note that the escape character is escaped.
   * @return A perl-like-formatted version of the regexp.
   */
  public String getOrdinaryNPRegEx() {
    return np.replaceAll("\\?:","");
  }

  /**
   * Get a perl-like version of the regexp used to match verb-phrases.
   * Note that the escape character is escaped.
   * @return A perl-like-formatted version of the regexp.
   */
  public String getOrdinaryVPRegEx() {
    return vp.replaceAll("\\?:","");
  }

  /**
   * Get a perl-like version of the regexp used to match pronoun-phrases.
   * Note that the escape character is escaped.
   * @return A perl-like-formatted version of the regexp.
   */
  public String getOrdinaryPPRegEx() {
    return pp.replaceAll("\\?:","");
  }

  /**
   * Get a perl-like version of the regexp used to match adjective-phrases.
   * Note that the escape character is escaped.
   * @return A perl-like-formatted version of the regexp.
   */
  public String getOrdinaryAPRegEx() {
    return adjp.replaceAll("\\?:","");
  }

  /**
   * Internal class used to represent a group-tag.
   */
  private class GroupTag implements Comparable {
    // tänk på att taggarnas nummer påverkar deras ordning när de sorteras!
    public static final int VG_BEGIN=1;
    public static final int ADJP_BEGIN=2;
    public static final int NG_BEGIN=3;
    public static final int PP_BEGIN=4;
    public static final int VG_END=5;
    public static final int ADJP_END=6;
    public static final int NG_END=7;
    public static final int PP_END=8;

    private int type;
    private int pos;
    public GroupTag(int t, int p) {
      type=t;
      pos=p;
    }
    public int compareTo(Object _tag) {
      GroupTag tag=(GroupTag)_tag;
      if(pos<tag.getPos()) { return -1; }
      if(pos>tag.getPos()) { return 1; }
      else {
        if(this.isOpening() && tag.isClosing()) { return 1; }
        else if(this.isClosing() && tag.isOpening()) { return -1; }
        else if(this.isOpening() && tag.isOpening() && this.type <
tag.getType()) { return  1; }
        else if(this.isOpening() && tag.isOpening() && this.type >
tag.getType()) { return -1; }
        else if(this.isClosing() && tag.isClosing() && this.type <
tag.getType()) { return -1; }
        else if(this.isClosing() && tag.isClosing() && this.type >
tag.getType()) { return  1; }
        else return 0;
      }
    }
    public int getPos() {return pos;}
    private int getType() {return type;}
    public boolean isOpening() { return (type==1 || type==2 || type==3 ||
type==4); }
    public boolean isClosing() { return !isOpening(); }
    public String toString() {
      switch (type) {
        case VG_BEGIN :
          return "<VG> ";
        case VG_END :
          return "</VG> ";
        case NG_BEGIN :
          return "<NG> ";
        case NG_END :
          return "</NG> ";
        case PP_BEGIN :
          return "<PG> ";
        case PP_END :
          return "</PG> ";
        case ADJP_BEGIN :
          return "<AG> ";
        case ADJP_END :
          return "</AG> ";
        default :
          return "";
      }
    }
  }
}
```

SUC1AFileReader

```
package groupdetection;
import java.io.*;
import java.util.regex.*;

/**
 * <p>Title: Noun and Verb group detector</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: Lund University</p>
 * @author Tommy Karlsson & Jimmy Andersson
 * @version 1.0
 */
public class SUC1AFileReader {
  /**
   * Reads a file on the SUC1A-format and returns the text with part-of-
speech-tags on the Granska-format.
   * All headers and other "non-sense" information is not present in the
resulting string. Also, all newlines are removed.
   *
   * @param f File-object refering to a file on the SUC1A-format.
   * @return A string with part-of-speech-tags on the Granska-format.
   */
  public static String read(File f) {
    String fs="";
    try {
      FileReader fr=new FileReader(f);
      char[] cv=new char[(int)f.length()];
      fr.read(cv);
      fr.close();
      String s=new String(cv);
      Pattern p;
      Matcher m;
      //remove all header/bogus lines
      s=s.replaceAll("(?m)^\\(\"<<.*?$[\\n\\r]*","");
      //remove all newlines
      s=s.replaceAll("(?m)[\\r\\n]+","");
      //a somewhat complex (at least hard to read) regexp, and unfortunately
not very efficient
```

```
      p=Pattern.compile("\\(\"<(.*?)>\"[^\\(]*\\(([^\"]*)\\s");
      m=p.matcher(s);
      while(m.find()) {
        fs+=m.group(1)+" ";
        fs+="<"+m.group(2).replaceAll("\\s",".").toLowerCase()+"> ";
      }
    }
    catch(Exception e) {System.out.println(e.getMessage());
e.printStackTrace();}
    return fs;
  }
}
```

TagRemover

```
package groupdetection;
import java.util.regex.*;
import java.util.*;

/**
 * <p>Title: Noun and Verb group detector</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: Lund University</p>
 * @author Tommy Karlsson & Jimmy Andersson
 * @version 1.0
 */

public class TagRemover {
  /**
   * Removes all group-tags from a string.
   *
   * @param s The string from which you want to remove the group tags.
   * @return The supplied string with all group tags removed.
   */
  public static String removeGroupTags(String s) {
    return
s.replaceAll("(?m)<((NG)|(/NG)|(VG)|(/VG)|(PG)|(/PG)|(AG)|(/AG))>\\s*","");
  }
  /**
   * Removes nestled group-tags from a string.
   * Note that this method is not guaranteed to be correct.
   *
   * @param s The string from which you want to remove the nestled group
tags.
   * @return The supplied string with all nestled group tags removed.
   */
  public String removeNestledGroupTags(String s) {
    Stack tag_stack = new Stack();
    Pattern p=Pattern.compile("<PG>[^<]*(<[/]?(?:(?:AG)|(?:NG))>)");
    Matcher m=p.matcher(s);
    while(m.find()) {
      tag_stack.push(new TagPos(m.start(1)-1,m.end(1)));
    }
    while(!tag_stack.isEmpty()) {
      TagPos tp=(TagPos)tag_stack.pop();
      s=s.substring(0,tp.getStart())+s.substring(tp.getEnd());
    }
    m=p.matcher(s);
    while(m.find()) {
      tag_stack.push(new TagPos(m.start(1)-1,m.end(1)));
    }
    while(!tag_stack.isEmpty()) {
      TagPos tp=(TagPos)tag_stack.pop();
      s=s.substring(0,tp.getStart())+s.substring(tp.getEnd());
    }
    m=p.matcher(s);
    while(m.find()) {
      tag_stack.push(new TagPos(m.start(1)-1,m.end(1)));
    }
    while(!tag_stack.isEmpty()) {
      TagPos tp=(TagPos)tag_stack.pop();
      s=s.substring(0,tp.getStart())+s.substring(tp.getEnd());
    }
    m=p.matcher(s);
    while(m.find()) {
      tag_stack.push(new TagPos(m.start(1)-1,m.end(1)));
    }
    while(!tag_stack.isEmpty()) {
      TagPos tp=(TagPos)tag_stack.pop();
      s=s.substring(0,tp.getStart())+s.substring(tp.getEnd());
    }
    p=Pattern.compile("<NG>[^<]*(<[/]?AG>)");
    m=p.matcher(s);
    while(m.find()) {
      tag_stack.push(new TagPos(m.start(1)-1,m.end(1)));
    }
    while(!tag_stack.isEmpty()) {
      TagPos tp=(TagPos)tag_stack.pop();
      s=s.substring(0,tp.getStart())+s.substring(tp.getEnd());
    }
    m=p.matcher(s);
    while(m.find()) {
      tag_stack.push(new TagPos(m.start(1)-1,m.end(1)));
    }
    while(!tag_stack.isEmpty()) {
      TagPos tp=(TagPos)tag_stack.pop();
      s=s.substring(0,tp.getStart())+s.substring(tp.getEnd());
    }
    return s;
  }

  /**
   * Removes all part-of-speech tags from a string.
   * Note that this method is based on the fact that all part-of-speech-tags
are all lowercase.
   *
   * @param s The string from which you want to remove the part-of-speech-
tags.
   * @return The supplied string with all part-of-speech-tags removed.
   */
  public static String removePartOfSpeechTags(String s) {
    return s.replaceAll("(?m)\\s<[a-z\\.\\-/&&[^ANPVG]]*?>","");
  }

  private class TagPos {
    int start;
    int end;
    public TagPos(int s, int e) {
      start=s;
      end=e;
    }
    public int getStart() {return start;}
    public int getEnd() {return end;}
  }
}
```

Ett gränssnitt med Naturligt språk till en TV programs-databas

Mikael Hallin
Department of Computer Science
Lund University
mikaelhallin@msn.com

Samuel Andersson
Department of Computer Science
Lund University
tetsu@algonet.se

Abstract

This report describes how we implemented a TV guide application that is controlled by natural language. The application is using simple strings for its input and output. The idea around the language processing part is to make it simple to implement. We are using keywords to parse the questions and the answers are then generated by templates.

1 Inledning

Denna rapport beskriver hur man kan applicera ett användargränssnitt med naturligt språk för att söka i en databas med information om TV-program. Projektet delades upp i tre logiska moduler, vilka är helt oberoende av varandra. De tre modulerna är en TV-programs-databas (TVDB), en språkbehandlingsmotor och ett GUI. Datan för TVDBn hämtas från olika webbplatser, beroende på vilket land som TV-programmen visas i. Språkbehandlingsmotorn är skriven i Java och har i sin tur egna små databaser med nyckelord. Slutligen skrevs ett enkelt GUI i Swing. Kommunikationen mellan de olika modulerna sker till stor del med en egendefinierad kommandosträng och programlistningar. Till och från slutanvändaren består kommunikationen av naturligt språk blandat med programlistningar.

2 Metoder

Vi började med att identifiera några nyckelfrågor som vi ville att programmet skulle kunna besvara. Exempel på frågor som vi tyckte var viktiga är "När börjar Simpsons?", "Går Seinfeld idag?" och "Går det några filmer ikväll?". Dessa frågor är ganska enkla, men det skulle visa sig senare att den strategin vi valt är väldigt modulär och klarar även mer komplicerade frågor som "Går det någon fotboll den här veckan på TV3?".

För att enkelt få tag på information om TV-program använder vi oss att ett befintligt projekt som heter XMLTV. XMLTV tillhandahåller diverse verktyg som är nyttiga för att hantera TV-tablåer. Projektet tillhandahåller även en hel del skript som kan ladda ner TV-tablåer via internet på ett tio-tal olika språk.

Eftersom programmet bygger på att man ska efterlikna en konversation med datorn, ville vi få GUIt att påminna om vanliga populära instant messaging klienter som t.ex. MSN Messenger och ICQ. Programmets huvudsakliga komponent är därför en stor textyta där konversationen hamnar, med växlande färger för användarens och datorns fraser. Nedanför denna yta finns ett fält där användaren kan skriva in sin fråga och en knapp för att "skicka", d.v.s. starta behandlingen av det användaren har sagt.

3 Implementation

Vi bestämde oss för att implementera projektet i Java, eftersom det går snabbt att jobba

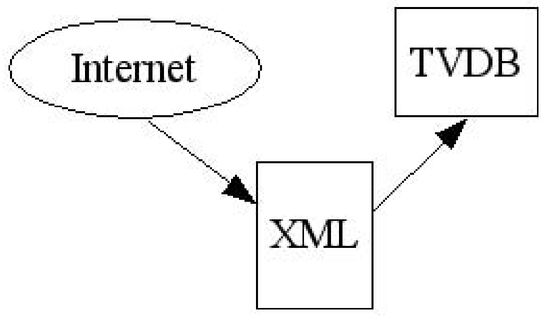


Figure 1: Datans väg från internet till vårt program.

med och har ett omfattande klassbibliotek. De delar vi hade störst nytta av var paketen rörande reguljära uttryck, XML parsning och Swing för att bygga GUIt.

Får att få tag på data som vi kunde använda för att testa programmet använde vi XMLTVs nerladdnings-skript. Vi använder oss av deras svenska skript som hämtar sin data ifrån DagensTV.

3.1 Databasen

Eftersom vi använder den mellanlagrade XML-datan får vi den stora fördelen att vårt program bara behöver förstå ett TV-tablå format. Detta innebär i sin tur att vi bara behöver skriva en parser som läser in datan. För att läsa in datan i minnet används Javas inbyggda SAX parser.

Datan lagras i en XML-fil, enligt en speciell mall, en s.k. Document Type Definition (DTD), som är specificerad av XMLTV-projektet. Mallen är ganska utförlig och kan förutom titel och beskrivning även innehålla information om vem som har regiserat och vilka skådespelare som är med i filmen.

3.1.1 Sökning

Databasen vi arbetar med innehåller programlistningar från cirka 15 kanaler en vecka fram i tiden. Detta motsvarar ungefär 3.000-4.000 poster i databasen, vilket får anses som en relativt liten databas. Vi tyckter därför inte att det är nödvändigt att implementera några avancerade algoritmer för indexering och sökning. Programmet som implementerades söker därför sekvensiellt och går igenom samtliga poster vid varje sökning.

Ett ”TV-dygn” är inte som ett vanligt dygn som slutar klockan 12 på natten. Tekniskt sett går ett program som börjar klockan 0:00 på natten på en annan veckodag, men vi ser det ofta som att programmet går sent på kvällen. Vi blev därför tvungna att manipulera datum och klocka för att klara av sökningar på dagar. Vi omdefinierade därför dygnet att gå mellan 04:00 - 04:00. På så vis hamnade program som börjar efter klockan 12 på natten på ”rätt” dag. Förutom sökningar på absoluta klockslag vill man även göra sökningar på mer abstrakta tidpunkter som t.ex. kväll eller eftermiddag. Uttrycket kväll definierades därför till ett intervall mellan 18:00-22:00.

3.1.2 Kategorier

Databasen vi fick tag på innehöll inte någon information om vilka kategorier som programmen tillhörde. Vi blev därför tvungna att själva skriva ett antal enkla regler för kategorisering av programmen. För att göra detta tittade vi i programmets beskrivning. I beskrivningen står det ofta saker som ”Amerikanskt action-drama från 1991”. Vi gjorde därför regler som flaggade programmen som filmer om den här meningen innehöll ord som komedi, thriller, action osv. Program som innehöll ishockey eller NHL, hamnade i hockey-kategorin.

Man kan sedan söka efter program kategorivis, till exempel all fotboll som sänds. Våra kategorier har även visst stöd för hierarkier. Tanken är att man ska kunna söka efter sport och sedan få upp alla sportrelaterade program. Vi hann dock inte bli klara med detta steg, utan nöjde oss med en platt struktur.

3.2 Språkbehandling

Vår plan med programmet är att det skulle klara av att både förstå och svara med naturligt språk. För att klara det på den korta tid vi hade för projektet, valde vi att göra en så enkel parsning som möjligt för texttolkningen. Vi valde därför en slags nyckelordssökning och bryr oss inte så mycket om grammatiken i meningarna. För utmatning gjorde vi färdiga mallar för svar på de olika typer av frågor som programmet stödjer. Mallarna fylls i med det som hittas i TVDB. Internt i programmet så kommunicerar de olika modulerna med varandra med hjälp av programlistningar och en egendefinierad kommandosträng, kallad Commandstring. Kommandosträngen är uppbyggd av informationen som parsas från frågan.

3.2.1 Nyckelord

För att få ut vad det egentligen frågas efter så hade vi idéen att man skulle leta efter nyckelord i meningen. Nyckelordet kan även vara en kort fras. Nyckelorden delas upp i olika grupper beroende på deras betydelse. De grupper vi har är

- Frågeord. Den här gruppen innehåller ord som oftast brukar finnas med i en fråga.
- Skådespelare. Här finns alla skådespelare, kan även använda smeknamn som nyckel.
- Kanaler. Alla kanalerna som finns med i TVDB.
- Dagar. Allt som har med dagar att göra, både namn och benämningar som ”idag”.
- Programtyper. Olika typer av program, som film och dokumentär.
- Tider. Klockslag och tidsbenämningar. T.ex ”eftermiddag” och ”ikväll”.
- Titlar. Alla titlar som finns i TVDB.

Varje grupp av nyckelord lagras i en egen fil, där varje nyckelord kopplas till ett standardiserat ord eller ett namn som skall användas internt i programmet. Genom att ha standariserade ord så kommer det vara likadant internt oberoende av vilket språk som används för inmatning. Filstrukturen är gjord så att det skall vara enkelt att stöjda flera olika språk, t.ex. för de svenska tidsorden

```
database/se/times.data
```

och såhär kan de första raderna i filen se ut

```
klockan (\d+:\d+) = #1
(\d+:\d+) = #1
ikväll = evening
i kväll = evening
kväll = evening
```

Det till vänster om likhetstecknet är nyckelorden och det till höger är de standardiserade orden eller namnen. Själva nyckelordet kan även vara ett regulärtuttryck. Att vi la till den funktionalliteten var för att vi skulle kunna skapa nyckelord som matchar mot vissa mönster. Mönstermatching kommer till nytta på ord som har en viss betydelse, men som kan se olika ut från fall till fall som t.ex. klockslag. Det till höger kan även referera till en regulärtuttrycksgrupp. Det görs genom att skriva # följt av gruppnummert. På det viset kan man få med saker i standardordet som har matchats i inmatningssträngen.

Efterhand som man letar nyckelord tar man bort alla träffar från inmatningssträngen. Att vi gör det är för att det inte skall bli två träffar på samma ställe. Så ordningen man väljer på nyckelorden har stor betydelse. Först ordningen på nyckelordsgrupperna och sedan kan det även ha betydelse på ordningen av nycklarna inom grupperna. För grupper valde vi att ha de först som representera namn, för att i t.ex. filmtitlar var det lätt hänt att de titlarna kunde innehålla ord som dagar och frågeord. Om man då råkat ta bort en del av en titel skulle man aldrig kunna hitta vilken film frågan gällde.

Efter all parsning har man fått ihop en mängd standardord och namn, som man sätter in i Commandstringen.

3.2.2 Commandstring

För att förenkla för sökmotorn till TVDB och för att få någon typ av standard som inte skulle vara så beroende på språk och formulering av frågor, skapade vi Commandstring. Commandstringen är uppdelad i olika segment, där varje segment i princip representerar en nyckelordsgrupp. Alla segmenten behöver inte vara ifyllda för att TVDB-sökmotorn skall kunna förstå vad den skall leta efter. Själva Commandstringen ser ut på följande sätt

```
command|time|day|channel|progtype|
title|actors
```

De flesta fält är mer eller mindre självförklarande, så command är den enda som kommer att beskrivas i detalj. Command är det segemnt som talar om vad det är för typ av fråga. Oftast så bestämms command genom en matchning av ett frågeord, men ibland så bestäms det beroende på vilka andra segment som är ifyllda eller inte. De commands som finns är

- asktime. Frågar vilken tid något går.
- askwho. Frågar vilka som är skådespelare eller värdar för en produktion.
- asktitle. Frågar titel på en produktion.
- nonsense. Om inte något vettigt kommando kunde hittas.

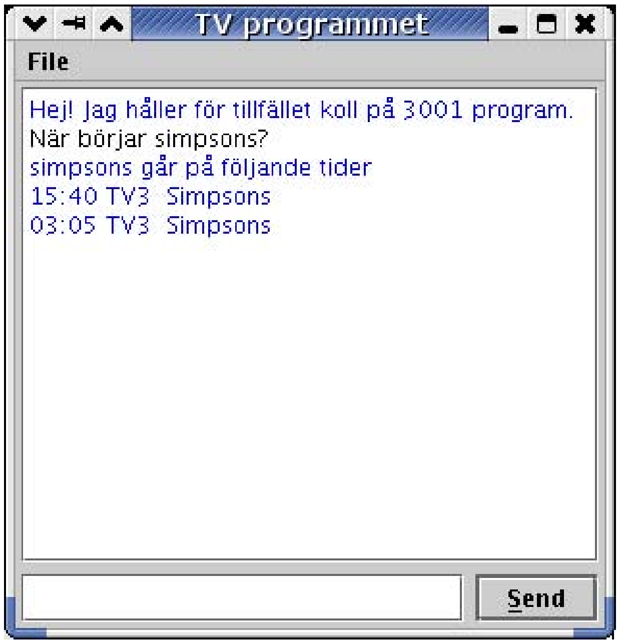
Några exempel på hur en Commandstring kan se ut

```
Går det någon komedi klockan 20:30
idag?
asktitle|20:30|today||comedy||

Vilken tid går seinfeld imorgon?
asktime||tomorrow|||seinfeld|
```

3.2.3 Mallar

För att generera svar till användaren på ett enkelt sätt valde vi att skapa ett mallsystem. På det viset skulle det inte krävas någon komplicerad kod för att skapa svar i formen av naturligt språk. Genom att dela upp mallarna i grupper, beroende på vad det är för typ av fråga man skall generera svar till, är det lätt att hämta rätt mall. Även mallarna ligger i filer, där filerna har samma logiska filstruktur som nyckelorden. Inom varje mallgrupp finns det tre huvudsakliga varianter av mallar. De tre varianterna är om man hittar många program, ett program eller inga program. Men



Hej! Jag håller för tillfället koll på 3001 program.
När börjar simpsons?
simpsons går på följande tider
15:40 TV3 Simpsons
03:05 TV3 Simpsons

Figure 2: Screenshot av GUIt.

där kan också finnas flera mallar för varje huvudvariant, som ser lite olika ut. Det för att man skall kunna få lite olika formuleringar på samma svar, bara för att få programmet att verka mer levande.

Alla mallarna börjar med en bokstav som talar om vilken huvudvariant som den tillhör. p för många program, s för ett program och n för inget program. Själva mallen är bara en vanlig mening där man stoppat in lite markörer på ställen där man vill att det skall stoppas in frågespecifika ord. T.ex.

```
s #title går #time på #channel
```

Vilket t.ex. kan producera svaret

```
seinfeld går 23:40 på ZTV
```

till användaren.

3.3 Användargränssnitt

Användargränssnittet är enkelt och intuitivt uppbyggt. Figur 2 visar ett screenshot av hur det ser ut när man kör programmet. I menyn kan man ändra språk på både in- och utmatning. Vi implementerade stöd för både svenska och engelska för att visa hur flexibelt systemet är.

4 Resultat

Prototypen vi fick fram var funktionell och användbar. Den klara av att parsa de testfrågor vi skapade i början av projektet och till vår glädje så klarar den även av mer komplicerade frågor utan vi behövde göra något utöver vad vi gjorde för att klara testfrågorna. Programmet klarar av att svara tillfredsställande på de flesta frågorna, men i enstaka fall kan svaren ibland bli lite felaktiga.

Vi hann även implementera så att programmet klarar av flera språk, ändringen för att göra det var inte stor. I princip var det att lägga till en meny och ändra lite i filstrukturen för att få plats för filer på flera språk.

5 Slutsatser

Valet vi gjorde med att ha nyckelord och databaser var lyckat. Det var enkelt att implementera och det är enkelt att lägga till nya nyckelord. Oftast när vi skulle utöka stöd för nya frågor så var det bara att ändra i databasen. Det kunde bli ganska avancerade förändringar bara genom att ändra i databasen, vilket var bekvämt då man slapp kompilera om hela tiden.

Mallarna var också ett lyckat drag, även om vi inte hann testa ut dem ordentligt på grund av tidsbrist, så fick vi de att fungera bra.

Referenser

DagensTV.com Website som har TV-tablåer. http://www.dagenstv.com/

XMLTV Projekt som har en mängd verktyg för att hantera TV-tablåer. http://xmltv.sourceforge.net/

Acknowledgement

Vi vill tacka Pierre Nugues för hans goda ideér under projektets gång.

Football Information Extraction System

ESTEVE LLOBERA
Lund University
e6801661@est.fib.upc.es

CIAN DALTON
Lund University
cian.dalton@student.cs.ucc.ie

JULIO ANGULO
Lund University
djjupa@canada.com

Abstract

We are going to make a program able to extract information from sportive texts. We will focus on football texts and the program will extract the winner and loser teams, as well as the final score and the location where the match has been played.

The program will deal with texts in two languages: english and spanish. The user has to tell to the program what language to use to analize the text because it will not try to identify the language.

1 Introduction

This system consists on an information extraction program that extracts relevant information from football related articles. Relevant information is considered to be the name of the participating teams, the resulting score of the match played and the location of the match. In other words, the program basically analyses the contents of the article and returns relevant information about the match.

In order to analyse the article the program follows some stages of development. At each stage the text is analyse in a different way, trying to obtain the correct result at the end. There are five stages of development, tokenizing the text, splitting each sentence, analysing each of these sentences, search for relevant patterns, look for important results, and finally display the results. These stages are explained in detail in this article.

The program will deal with texts in two languages: English and Spanish. The user has to tell to the program what language to use to analyze the text because it will not try to identify the language. The default language is set to English, in case the user fails to choose the language of the article to be analyzed

2 Program Implementation

The project has been made using a design in layers. It has 5 layers (see figure X): tokenizer, sentence splitter, phrase analyzer, pattern searcher and table analyzer. We will make an small overview of all the stages now, and they will be explained in more detail later. Each layer generates an intermediate representation of the analyzed text. The tokenizer returns an array with the tokenized text, the sentence splitter splits the tokenized text in sentences, which are just an array of tokenized words. The phrase analyzer is called for each sentence found, and returns an instance of a class Phrase for each one. The pattern searcher gets the entire Phrase objects and search for sportive patterns that can provide information about a football match. If a sentence contains relevant information, it is stored in a class called Match, which contains winner, loser, score and location. Of course, most of those fields will be filled in empty because a sentence usually doesn't contain all that information. For each sentence with useful information the pattern searcher will create a new Match object with that information. Finally, the pattern searcher returns a table with all those Match objects, and the table analyzer will try to merge some of those "matches" in order to get all the possible information (score, location, etc.) and to return the real match.

2.1 The Tokenizer

The first step on processing the text is tokenizing it. The program uses Java's tokenizing mechanism for achieving this task easily. The class *StringTokenizer* returns each token that is found in the text. This particular program also considers the following characters as tokens: '!', '/', '?', ':', '-', ';', '¿', '\n', '\t', '.', ',', '(', ')'. Tokenizing the text makes it easier for analysing each word and each sentence in the later stages. This is a very simple step that it is very easily performed with the help of Java classes.

At this stage the program already considers the difference in language. We found that it was useful to recognize the accented letters at the beginning of the text and transformed them into a not accented letter. In Spanish all the vowels can be accented (á, é, í, ó, ú) and this can bring complications at later stages. In general, if there is a word in the text, which contains an accented letter, this letter gets transform to its non-accented equivalent.

2.2 Sentence Splitter

The constructor for this class takes two parameters, the input (String[]) and a Boolean value indicating the language of the input text, either English or Spanish.
The class has two major methods, *thereIsSentence*, which returns a Boolean value indicating whether or not there are any sentences left in the input text, and *nextSentence* which returns the next sentence in the input array. The program uses a variable *startIndex* to indicate the index position where each sentence starts. This index is incremented with each step of the process and was critical to both of the methods.
The method *thereIsSentence()* simply returns true if the length of the input was greater than the start Index, i.e. the *startIndex* has not reached the end of the input array and therefore there is still at least one more sentence to split.

The method *nextSentence()* uses a loop to go through the input array and firstly checks if each character equaled a sentence delimiter (".", "?", ",", "!", ";" or ":") and considered all characters between the two delimiters a sentence by adding the sentence to our list (LinkedList) and incrementing the *startIndex* until it is greater than the index of the delimiter so we can move on and find the next sentence. The process then repeats until the end of the input is reached and thus all sentences are split or separated. Each LinkedList *nextSentence* is converted to an array and returned. The list that is returned, and which is supposed to contain one sentence of the text, is furthered analyzed by the *PhraseAnalyzer* class.

2.3 Phrase Analizer

The phrase analyzer is called for each sentence in the text, so it doesn't have any knowledge about the sentences before. We would have liked to have a good phrase analyzer, but it had increased too much the complexity of this layer and besides, it's not the main objective of this project. So we decided to use a simple approach to a full phrase analyzer. For each language (English and Spanish) we keep a list of verbs, prepositions and conjunctions. The idea is to be reading the sentence until find one of those, that we have called, bounding words. Each group of words between two of those bounding words is a group in the sentence (the subject, the verb, the object or any prepositional sentence). The algorithm is the following: we consider the type of a group depending on the first word of the group. What we find until the first bounding word, is always considered to be the subject of the sentence. Once we find a bounding word, if it is a verb word, this will be the verb of the sentence. In this case, we start reading the words after it. While the word is a verb word, we attach that word to the verb of the sentence. This way we deal with sentences with a composed verb. If the word is a preposition, we consider it a prepositional phrase, and if it's not a preposition, we consider it as the object of the sentence. A conjunction is only used to separate groups.

If we find more than one verb or direct object, only the first one is used. The other ones are ignored.

So, if we analyze the next sentence:

10 November 2003 Blackburn Rovers FC ended a run of five consecutive
Premiership defeats with a nervy 2-1 win against Everton FC at
Ewood Park tonight.

we get:

```
 -  Subject:10 November 2003
     Blackburn Rovers FC ended a
     run
 -  Verb: win
 -  Object: [NULL]
 -  PP: of five consecutive Pre-
     miership
     with a nervy 2 - 1
     against Everton FC
     at Ewood Park tonight
```

ended is not in the verb list so it's not considered a verb.

This information is stored in a Phrase object, which has the following structure:

```
public class Phrase{
  public Group subject;
  public Group verb;
  public Group object;
  public Group[] pp;

        ...
}

class Group{
 public static final int NOUN = 0;
 public static final int VERB = 1;
 public static final int PREP = 2;
 public static final int ADVERB = 3;
 public static final int CONJUN = 4;

 public int groupClass;
 public String[] data;

        ...
}
```

The phrase object will represent the analyzed phrase, and that's what the phrase analyzer gives to the pattern searcher.

2.4 Pattern Search

After the phrase is analysed and divided into subject, verb, object, and prepositions the program searches for relevant patterns on each of these fields. To do this the program uses a super class PatternSearch and two subclasses that extend it PatternSearchEnglish and PatternSearch-Spanish in order to deal with the difference in language. These subclasses basically consist of a list of words or sentences that are likely to appear in the different fields, we call them *patterns*. There are patterns related to relevant verbs, to relevant words, to locations, to prepositions, and to scores. For example, some relevant patterns concerning the name of a team were coded in java as an array of patterns elements (i.e. an array of type Pattern):

```
Pattern[] wordPatternSpa = {
 Pattern.compile("[A-Z]\\w*"),
 Pattern.compile("[A-Z]\\w* [A-Z]
                  [A-Z][A-Z]*"),
  . . .
}
```

where the first element of the array will identify all words starting with Capital letter, the second element will identify all words starting with capital letter followed by a space and at least two capital letters (or initials), therefore it will recognize words like ‘Barcelona FC’ or ‘Barcelona FFC’.

At this stage the program takes a big step in determining the winner and the loser of a match, as well as the score and the location. To determine the winner and loser it considers five types of verbs that can be relevant to a football article, verbs about winning and losing in the active form, about winning and loosing in the passive form, and about drawing. In general, depending on the type of verb, either the sentence before or after it is considered to contain the name of either the winner or the looser. For example, if the phrase being analyzed at the moment contains a winning verb in the passive (e.g. *has been won*) form then the program assumes that it is likely to find the name of the winner after the verb. In the other hand, if the phrase has a winning verb in the active form (e.g. *triumphed*) then the program considers the winner to be found before the verb and therefore the user after the verb. This doesn't mean that the final output of the program depends on finding one of these verbs and concluding we can find the appropriate winner and looser, because there are many phrases to be analysed and the results of all of them will be compared at a later stage.

To determine the location of the match, the program considers sentences or groups containing only two prepositions related to location *'at'* and *'in'* (in Spanish there is only one preposition *'en'*. Reading over some corpus we found that it is only after this two words that the location of the match is mentioned in an article. We can encounter sentences as *The match was won by Barcelona in Camp Nou* where it is very simple to detect the location. In the other hand an article might have the sentence *The match was won by Barcelona in September*, for this reason we decided to have a list of words that the program might misinterpret as important, we called this list the unwanted patterns and it includes names of months, days of the week, capitalized prepositions, etc. and it helps the program to get rid of words that are not likely to appear in the final result. A text can also contain the sentence *The team won the game at the beautiful city of Bristol* and again by taking only the relevant patterns the program gets rid of the unwanted *the beautiful city of* to conclude that the relevant location is *Bristol*.

The process of determining the scores is slightly different. First the program tokenizes each of the groups in the phrase to look for a pattern of the form *Number Union Number*, where *Number* is any integer and *Union* is a word or character that appears between the two numbers which is commonly use to display a result, such as '-', 'to', 'against', 'by', ':'. If a pattern of this type is found, then the program considers these two numbers to be a possible score. However, the numbers found can happen to be only a partial result or the result of another match. We found a big difficulty in finding the correct final score when the text presents more than one on these *Number Union Number* patterns. For this reason we implemented the program so that it assumes that the pattern with the highest integer numbers is indeed the final score of the match.

All the results found by the *Pattern-Search* class are considered possible results. These results are stores in a list of *Matches*, which is analyzed at the next stage of the development process of the program.

2.5 Table Analyzer

The pattern searcher yields a table that contains many Match objects. As we already know, each Match object represent a match, or part of the information of a match. The objective of this layer is to merge all the matches, so we can get a match with all the information given in the text. The reason for this is because, as we analyze sentence per sentence, each Match object created only contains the information about the match that appears in that sentence. But in most of the cases, the information of a match is given in multiple sentences. For example you can say the team A won team B in one sentence but the score can be in another one. For this reason, the table analyzer will merge all the matches together, in all the possible combinations. Two matches will be merged only if there is not contradiction when merging them.

The table analyzer works in 2 steps. First it gets all the team names it finds on all the matches. Second, it merges the matches considering the team names found.

When the pattern searcher creates a Match object, it doesn't take care about the team names. It only knows that the subject or the object of the sentence may refer to a team name, and fills in the fields with that information. So when we get two matches, for example one with a winner called "The great FC Barcelona team" and another one with a winner "The Barcelona team", we must tell the system that those two teams refer to the same team: Barcelona. That's the objective of the first step in the table analyzer layer. Once we have all the team names that appear in the text, it is easier to know when two matches can be merged using the real team name and not all the expression.

The method we have used to get the team names is quite intuitive: we look for team patterns in all the winner and loser field of all the Match objects. There are not many teams' patterns, but the problem is that it's very difficult to distinguish team names from football players. For example FC Barcelona is a team, but Javier Saviola is a football player, and both will be considered as team names.

Once we have the team names, the program starts to merge all the possible matches. Two matches will be merged if no contradiction appears when merging them. Contradictions are the winner and the loser would be the same team, different score or location, or different winner or loser. For example if a match with a winner Barcelona and a match with a winner Madrid cannot be merged. But a match with the winner null and the loser Madrid and another match with the winner Barcelona and the loser null can be merged. The resulting match will have winner Barcelona and loser Madrid. This resulting match will be added to the list of matches, so it can be used to be merged again.

When we have merged all the matches, we need to select the match the text is talking about. For that reason, each match object is augmented with a counter. When the pattern searcher creates a match, if the fields winner and loser are both null, the counter is 1, if either the winner or the loser is not null, the counter is 2, and if both winner and loser are not null, the counter is 3. When two matches are merged, the counter of the resulting match will be the addition of the two original counters. At the end, we return the match with the higher counter.

3 Measures of Performance

We were confident that the precision and recall of the program in general were fairly good. We tested the system with approximately 30 Internet articles narrating football matches and we estimated a recall and precision between approximately 40% and 60%. In the case when one article mentioned more that one match this percentage of recall and precision dropped considerably. Also, when the text was extremely large, the displayed result was usually not the correct one, and therefore precision and recall also were reduced by approximately 30%, which is a big decrease in performance.

Error handling was not developed in detailed, however we assure that the program never terminates abruptly for any reason. If the input is not correct and the language is selected badly the program might display merely garbage (we said earlier that the program is not concern with checking the language of the input text), but it will not create exceptions, freeze or terminate. Instead, the user is given a chance to clear the text area and try analyzing the text again.

4 Conclusions, Observations and Future improvements

From developing this program we found many interesting facts and difficulties about processing languages with machine instructions. We also realized about the differences and similarities in processing two distinct languages, but fortunately we found it easy to separate the implementation of the two.

There are many things that could have improved in the program. The following is a list of some important observations and future improvements that we were aware of, but because lack of time couldn't be implemented in this version.

1. A better phrase analyzer. A bad one accumulates too much garbage for the *TableAnalyzer* to analyze. Statistically the names of the teams the match talks about are more likely to appear more times in the text, therefore the counter should be enough to get the correct name of the teams, but in the real life it's not so. This could definitely be improved by removing the garbage generated due to the bad phrase analyzer.
2. Optimally the analyzed text only talks about one match, however texts relating multiple matches can also be analyzed. The match with more references made on the text should be the final result, although the program will not always return this as the correct result. In general

the program performs much better if the article only talks about one football, both precision and recall are higher in this case.

3. An improvement in the user interface is the liberty for the user to choose a text file located in his machine, instead of having to copy/paste the article into the text field.
4. The order of the patterns in the array of patterns could be sorted in a way that the most likely pattern appears in the first index of the array, the second most important in the second index and so on. In this way the performance of the system will improve since it doesn't have to go through many elements of the array. In other words, we could have carried out a statistical study to find out the likelihood of occurrence of each of the patterns. In this way we can place the most likely occurring patterns at the beginning of the array of patterns so that the program doesn't have to go through the whole array.
5. Java was a useful tool in the development of the program, since it has build in classes that are very useful for language processing, such as string and stream tokenizer and a Pearl like pattern functionality.
6. The differences in the language were handled by having subclasses for each language that extended a superclass with the majority of the functionality. The subclasses included mostly the relevant vocabulary that can be found in the two different languages.
7. The handling of errors was not done with too much care. Handling all exceptions and possible technical errors could make the system more robust and trustful. However, we made sure that the program never crashes, as mention above.

Like every software development process, there are still some things that we could have done better, but because lack of time, tools and some knowledge we were not able to implement it as good as we would like to. However we were very satisfied with the final result of the system since the accuracy and precision were beyond our expectations

References

Pierre Nugues, 2003. *Assignment #2: Information Extraction.* http://www.cs.lth.se/Education/Courses/EDA171/cw2.html

Pierre Nugues, 2003. *Corpus Processing Tools*

Erik Lindvall and Johan Nilsson. *Extracting information from Sport Articles in Swedish using Pattern Recognition*. Lund University

Language Detection based on Unigram Analysis and Decision Trees

Sofia Bastrup
LTH Lund, Sweden
`dat00sob@ludat.lth.se`

Christina Pöpper
ETH Zürich, Switzerland
`poepperc@student.ethz.ch`

Abstract

This document describes the process of implementing a decision tree for language detection. First, text profiles are computed for each text in the training set and out of this a decision tree is built. Finally evaluation is made with query texts. To compare the performance of different approaches, a few variations of trees are implemented; sets of 26 respectively 56 characters as attributes are compared, furthermore a 'direct-child'-tree versus a 'neighbours'-tree are implemented for comparison. The 'direct-child' tree gave the greatest number of correct answers, in contrast to the 'neighbours' tree that did not give any incorrect answers (however many no results). Suggestions of enhancements of performance is given. The authors conclude that the language detector gives comparatively good results given that the implementation only considers unigrams.

1 Introduction

The goal of this project was to write a *language detector*, i.e. a system to identify the language of a given text automatically - out of a predefined number of possible languages.

Applications of automatic language detection involve language processing such as automatic retrieval of texts in the desired language (e.g. from the world wide web) as well as studies of language use (e.g. estimation of language frequencies in the internet).

Several broad approaches exist to the problem of language detection.

One obvious technique is to use a lexicon for each possible language and compare the words in the sample text with those in the lexica to find the lexicon with the highest correlation. This involves huge numbers of data to be managed and processed as well as difficulties when dealing with highly inflected languages. On account of the drawbacks of the lexicon method, grammatical words are used as discriminant in [Gi2].

Another technique is to use the alphabet. But, as stated in [Gi1], the alphabet is not very useful, because accented characters are not as frequent as needed and they often belong to several alphabets. Thus, this clue does not really allow to discriminate the right language.

A further approach is the use of n-gram analysis. *n-grams* are sequences of n letters. The basic idea is to compute, from a training set, a profile for each considered language based on the probability of letter sequences. For a given text the language with the nearest profile is selected.

On account of the drawbacks of the lexicon and alphabet methods, we decided to implement the last of the mentioned techniques: *language detection by n-gram analysis*.

2 Decision Tree Approach

The process of language detection can – in our case – be divided into three steps. In the first step, the frequencies and probabilities of the letters (unigrams) and combinations of letters (n-grams) of the training set are computed. This gives a profile for each language and has do be done only once. As the second step, we decided to build up a decision tree out of these profiles. Finally, the third step is to evaluate the decision tree for a given text.

In our implementation we only consider unigrams (i.e. letters) so far.

2.1 Training set

The implemented language detection method depends entirely on the quality of the training set. For obtaining the training set we used the Europarl Corpus of European Parliament Proceedings ([EPP]), version 1, for the following ten languages: *danish, dutch, english, finnish, french, german, italian, portuguese, spanish,* and *swedish,* each consisting of texts with 17-22 million words, corresponding to 60-80 MB each.

2.2 Language/text profile

The profile for each text in the training set (corresponding to a certain language) as well as the profile for the query text is given by the probabilities of all n-grams (in our implementation: unigrams). In order to deal with these probabilities and to transform the training set profiles into a decision tree, the probabilities are converted into intervals. Eight intervals were used corresponding to the probabilities of letters:

interval	corresponding probability	interval	corresponding probability
0	0 - 0.001	4	0.09 - 0.12
1	0.001 - 0.03	5	0.12 - 0.15
2	0.03 - 0.06	6	0.15 - 0.18
3	0.06 - 0.09	7	> 0.18

Interval 0 is not only assigned to letters not occuring at all, but also to letters occurring with a tiny probability of smaller than 0.001 (0.1%) in order to take loan words from different languages into account and prevent these from contaminating the profile of a language.

After this step, a typical profile for a language looks like this example for danish (containing only the common character set, see also section 2.3.4): [letter:interval]

a:2	b:1	c:1	d:2	e:6	f:1	g:2	h:1
i:3	j:1	k:2	l:2	m:2	n:3	o:2	p:1
q:0	r:4	s:2	t:3	u:1	v:1	w:0	x:0
y:1	z:0						

2.3 The decision tree

2.3.1 General building and structure

A decision tree is used for making classifications. The input for the tree is given by examples consisting of values of the set of attributes. Each internal node represents a test of the attribute value. The branches of the node are labeled with the different interval values. When reaching a leaf node the classification of the example is returned.

In our implementation of the decision tree the examples are represented by texts of different languages - a set of training texts when building the tree and a set of test texts to evaluate the tree. The classification attribute is the language of the text.

The attributes are the text profiles that are calculated out of every text in the training set. The values of the attributes are the intervals which correspond to the probabilities of the unigrams.

2.3.2 The implemented trees

Two different implementations are used in the language detector. We call them *direct-child-tree* and *neighbours-tree*.

Direct-child-tree: For the direct-child-tree, the idea is to take the profiles of the training texts and assign each of them exactly one path through the tree. In the final leaf, the language of the text is stored. I.e. for every internal node, containing a letter (the chosen attribute), the interval corresponding to this letter in the training text gives the child were we have to go on recursively.

In a first version of an implementation we only had one example for each language in the training set. This tree had a lot of leaves where no language could be determined. This happened because the query text has to have a profile much as same as the training text to be detected (in a path where all the attributes have to be used, the two profiles have to be identical). To overcome this problem we split up the training text into about 300 smaller text for each language. Then a query text has a higher probability of having a similar profile as one of the 300 training text profiles.

Neighbours-tree: Besides the direct-child-tree described in the previous paragraphs, we implemented a second approach, the *neighbours-tree*. The determining motive behind this approach is the fact that the direct-child tree (in its first version) is strict in respect to the profiles: the intervals of the n-grams of a query text has to correspond to the profile of a training text for this language. As described, the more training texts for each language are used for the direct-child-tree, the greater is the probability that a similar profile exists. But the problem remain, especially when the intervals are small and the probabilities lie at the border of the intervals (e.g. letter *a* may have probability 0.089 in the training text corresponding to interval 3, in the query text the probability may be 0.091 corresponding to interval 4, but the languages may be easily identical).

In the neighbours-tree we compensate for this by using only the profile of one text (of 60-80 MB size; concatenation of all texts of the same language used for the direct-child-tree) for each language and by assigning the language to the child of the interval as well as to both of the child's neighbours. Thus, texts varying only slightly in the intervals may be recognized as the same language nevertheless.

In Figure 1, a typical node in a neighbours-tree is depicted. Let's say, the languages danish and finnish are still possible and attribute *e* is chosen. The interval for *e* is 6 for danish and 4 for finnish. Then this part of the tree will look like this figure. Only for interval 5 both languages are still possible.

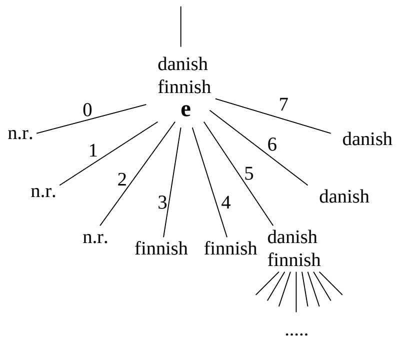


Figure 1: Typical node with children in the neighbours-tree. n.r. = no result. Letter e is chosen as attribute for this node.

The drawback of this approach are the many overlappings of letter intervals for the languages. The tree gets very big, because for many branches a great number of recursive steps has to be made. Because of limited memory, we had to limit the height of the tree to 8.

2.3.3 Choose attribute

When building a decision tree an appropriate attribute has to be chosen in each internal node. The method for choosing the attribute (we call it 'choose attribute method') is of importance concerning the depth and effectiveness of the tree. A "perfect" attribute is one that splits the set of examples into new sets in which all examples have the same classification. Choosing the right attribute gives a compact tree with a smaller search depth.

Two different choose attribute methods were implemented.

In the 'direct-child'-tree, information theory is used. The goal is to choose the attribute as similar as possible to the "perfect" attribute. To choose an attribute calculations of how much information is gained are made for all the attributes. This is compared to how much information there is currently. When building the tree the information content is calculated by counting how many examples there are in the set of each classification of the goal attribute. In case a node is reached when there are no more attributes in the set to choose from, but there are still examples not having the same classification, then a majority value is determined. This means that the language having the greatest number of examples in the set is returned as the classification of the text.

When building the 'neighbours'-tree there is only one example for each language (see section 3.2). Therefore, a simple choose attribute method is enough. Among the examples it calculates the difference between the highest and the lowest value for each attribute. The attribute with the largest difference is chosen.

2.3.4 Variation

Special characters: In our first approach we only considered 26 characters of the alphabet, the letters that all languages included in our research have in common. To develop our program further we extended the set of attributes to 56 different unigrams (containing letters such as $\hat{a}$, $\grave{e}$ and $\acute{o}$). This set now includes all special characters that exist in the considered european languages.

3 Results

Our test set consists of 33 different texts in the ten considered languages - at least three texts for each language. The (mainly contemporary) test texts were obtained from miscellaneous sources, from newspaper articles to scientific reports and literature. The sizes vary between about 270 and 67,600 words (or from 1.7 to 430 KB). To our surprise, we could not detect a correlation between the size of the text in this range and the correctness of the result. We suppose, that this correlation requires a greater number of test.

In the following, the reporting of the results is split up into the two implemented kinds of trees.

3.1 The 'direct-child-tree'

For the direct-child-tree each branch is labeled only with one value of the attribute. This in comparison with the 'neighbour-tree', where each branch is assigned three adjacent values.

Our results are as follows (33 tests):

tree	correct	false	no result
1	24	5	4
2	22	9	2
3	24	6	3
4	24	7	2

Four slightly different trees were implemented to compare the results. The first two compared the choose attribute methods. The method that uses information theory (tree 1) only appeared to be slightly better than the simple method (tree 2). This improved method gave two more correct answers and four fewer incorrect answers.

The second variation in the 'direct-child' approach is the tree with 26 attributes respectively the one where also the special characters are included as attributes. Both trees use the information theory when choosing attribute. Comparing those two trees (tree 1 and 3) showed surprisingly not any big difference at all in performance of classification. They both gave 24 correct answers. Furthermore the 56-characters-tree actually gave one more incorrect answer, which makes the 26-characters-tree the tree that showed best performance in the research. Still there was such small difference between those trees that it is reasonable to assume that in a more extensive research, those trees would yield the same performance.

Tree 4 used 56 characters and the simple choose attribute method.

All four trees determine a majority value when there are no more attributes.

3.2 The 'neighbours-tree'

The recursion depth for our version of the neighbours-tree is 8, bounded above by the size of memory on the used computer with a limited student account. Due to the height 8 of the tree (the maximal height is 26 or 56 respectively depending on the number of considered characters), many branches do not contain a result in the final leaf, as several languages are still possible, but can not be determined any further. Hence, we got many 'no result'.

We tested the neighbours-tree for the 26 character set and our first chose attribute method (a majority language would make no sense here as there is only one text for each language).

Our results are as follows (33 tests):

correct	false	no result
5	0	28

What strikes is the fact that there are no false answers - either the result is correct or there is no result at all.

More than likely, the number of 'no results' would decrease if the number of recursion steps could be increased.

4 Enhancements and Alternatives

An obvious enhancement is to consider n-grams, i.e. combinations of letters, and not only unigrams, because n-grams potentially contain much more information about the language. Many languages have bi- and trigrams that are representative for them, such as the english '*th*' or the german '*sch*'.

An alternative to the implemented decision tree approach is to use a *vector approach*. Thus, each language would be assigned a vector with 26 or more elements, depending on the number of characters taken into account. The result of the language detection will be the language whose vector has the smallest distance to the vector of the given text. The definition of distance depends on the chosen method. We did not implement this approach and thus can not compare the decision tree and vector approaches.

One more alternative turns up: Would the language detector show any improvement in performance if the value zero would represent no occurrences at all of a character in the text, instead of the interval 0 - 0.001?

There is obviously a difference between the letters that exist in a language only rarely (at few occasions) and the letters that do not exist at all. As for example the letters ä and ö in Swedish that do not appear in any latin language. Those characters can give a unique profile to the language they appear in. Still, no big difference in performance was detected between the 26- and 56-characters-trees. One way to make a distinction between these trees would be to represent only no occurrences with the value zero. There comes though a risk with this approach. The letters that exist only rarely in the language, could in one specific text be assigned the value 0 because in this text there are no occurrences at all. This is the reason why we chose to assign the value zero to those letters with a probability of 0 - 0.001. Anyway some emphasis on special characters would with high probability yield better performance.

5 Conclusion

A surprise for us during the development of the language detector was the fact that the enhancement from the simple character set to special characters did not yield the wished improvement. A reason for this may be that the special characters are not treated in any specific way.

Still far from a complete and satisfying program we were nevertheless surprised by the comparatively good results yielded by the very simply approach with unigrams. We are confident that an extension to two- and three-grams will improve the prediction significantly.

Acknowledgements

The authors gratefully acknowledge the very useful suggestions and help for this work by Pierre Nugues and Richard Johansson.

References

[CT] William B. Cavnar & John M. Trenkle. *N-Gram-Based Text Categorization*. Environmental Research Institute of Michigan.

[EPP] Europarl Corpus of European Parliament Proceedings. *http://www.isi.edu/ koehn/europarl*.

[Gi1] Emmanuel Giguet. *Categorization according to Language: A step toward combining Linguistic Knowledge and Statistic Learning*. Université de Caen, France.

[Gi2] Emmanuel Giguet. *Multilingual Sentence Categorization according to Language*. Université de Caen, France.

[GN] Gregory Grefenstette & Julien Nioche. *Estimation of English and non-English Langage Use on the WWW*. Xerox Research Centre Europe, France.

Why use buttons when natural language dialogue makes interaction easier: the Winamp Project

André Hellström, Nabil Benhadj, & Johan Windmark
Lund University, 2003

Abstract

This project was intended to show the possibility of using natural language dialogue with standard software in a typical PC environment. The prototype system integrates spoken language with the Winamp media player. As a result of this Winamp will be totally controlled by spoken English.

1 Introduction

The aim of the project is to create a spoken agent to control Winamp. Winamp is a leading media player that is easy to manipulate and control, making it well fit for the purpose. The interaction between the user and the agent is through voice recognition and speech feedback. The system should be easy to understand, even for a first time user. Another goal is to make the system robust enough so that it responds correctly with a high probability.

2 System overview

The system code is implemented in C++ using Microsoft Speech SDK (SAPI 5.1). The tutorial coffee0 that came with the SDK was the starting point of the project and was expanded to contain the state machine described below. Microsoft Speech can work in two modes: dictation mode and command mode. In the dictation mode, whole sentences are parsed. In dictation mode, the possible utterances are predefined in a grammar and each utterance is matched to a specific rule. The grammar for the rules is defined in an XML-file.

3 Implementation

The system code is implemented in C++ using Microsoft Speech SDK (SAPI 5.1). The tutorial coffee0 that came with the SDK was the starting point of the project and was expanded to contain the state machine described below. Microsoft Speech can work in two modes: dictation mode and command mode. In the dictation mode, whole sentences are parsed. In dictation mode, the possible utterances are predefined in a grammar and each utterance is matched to a specific rule. The grammar for the rules is defined in an XML-file.

4 State-machine

The flow of the system is:

1. The user gives a command to the system.
2. The system gives feedback on the command
3. Possibly gives a command to Winamp.
4. Go back to 1

To constrain the number of different commands available to the user at a given time, a state machine is used.

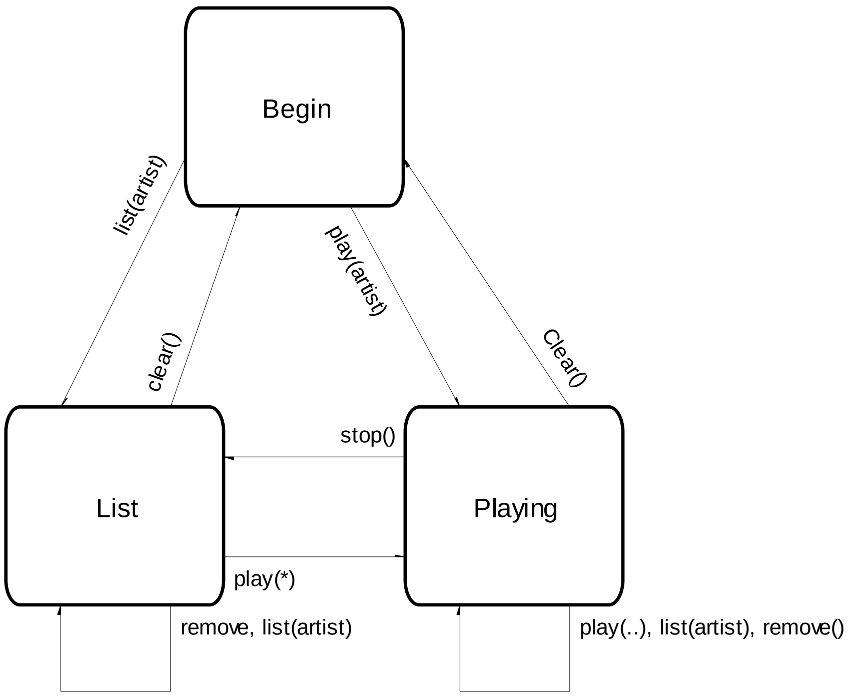


Figure 1

The state machine uses following concepts:

States, i.e. Begin, Playing
Transitions, i.e. Begin->Playing
Actions, i.e. play(Artist)
Rules, i.e. Rule7 triggered by ”* Play Madonna”

Let’s start by looking at the message handling code, suppose we’re in state Begin and have received a message from rule7.

```
messageHandler(message)
 switch(STATE)
  case Begin:
    switch(message)
      case RULE1:
        transitionBeginPlaying1(message);
      case RULE7:
        transitionBeginPlaying2(message);
      case RULE3:
      case RULE2:
        transitionXZ1(message);
      default:
        ”Error: Unhandled rule!”
  case Playing:
    switch(message)…
      case RULE7:
        transitionYY(message);
      default:
        ”Error: Unknown state!”
```

Depending on the state, different transitions can be called for the same rule, that is, the same rule can trigger different transitions. Also note that different rules can call the same transitions, within the same state. Let’s look at the transition code:

```
transitionXY2(message):
end_X()
visual and/or audio feedback code()
action_play(message.argument1)
begin_Y()
```

First a stop code is called for the state X. It will turn off all rules. Then some feedback is given to the user and the appropriate action is called. Finally the start code for Y is called. It will turn on all the Y-rules, the same rules that can be caught in the message handler under state Y, and set the current state to Y. In this case action_play(...) means load a Madonna playlist and start playing it, as a command to Winamp.

To summarize: The active rules depend on the current state. When a rule is triggered by the speech recognition system, it’s triggering a transition. The transition calls the appropriate action and changes the state, thereby changing the set of active rules.

5 Grammar and rules

The speech recognition system divides the input into phrases, where each phrase is surrounded by silence. Consider the case that a playlist is loaded and the user wants to hear a specific song. The rule to catch this looks like this:

```
<RULE ID="VID_PlayNumber"
TOPLEVEL="ACTIVE">
<O>
    <L>
        <P>could you</P>
        <P>will you</P>
        <P>I want to</P>
        <P>the song I want to hear</P>
        <P>I need to</P>
    </L>
</O>
<O>please</O>
<O>
    <L>
        <P>Play</P>
        <P>Start</P>
        <P>Hear</P>
    </L>
</O>
<O>
    <L>
        <P>song</P>
        <P>piece</P>
        <P>track</P>
        <P>tune</P>
    </L>
</O>
<RULEREF REFID="VID_Number"/>
</RULE>
```

The most important words for this rule is the number, last in the grammar. VID_Number, here defines one of the numbers 1 to 10 and is the sub rule that actually activates the VID_PlayNumber rule. The rest of the rule is only a list of optional padding so that the rule is triggered not only by an utterance like “Number one” but also by “Could you please play track number one”. The example pretty well catches the structure of all rules.

6 Future possibilities

Some interesting and important details of improvement would be to make the program more stable and increase flexibility. Below are a couple of suggestions of how this could be accomplished.

An easy way of making to program more dynamic is to provide an easy tool for automatically adding new artists and songs to the play list. One way of doing this is by having a parser read all music files in the designated catalogue and then adds whatever artist/song is missing in the XML-file each time the program is executed. Some system parameters would then have to be added into a text file for the program to read at start up.

Further improvement on the voice recognition would be preferable to increase stability of the speech-to-text input. A first and an easy way of enhancing the voice recognition input in the interface would be by providing a high quality microphone or possibly a headset.

Other possibilities would be to make a system that works as a speech user interface for programs in general. The only change required would be to change the grammar file (XML-file).

7 Evaluation

The system was tested continuously during development. After each test, the grammar and state machine was altered to enhance performance. The problems encountered were that user commands were not recognised or that rules were triggered without proper reason. In the final state of the project, a user knowing the system architecture can go through all actions with very few incorrect system responses. The problem with rules triggered without the correct corresponding user input still remains.

8 Conclusion

The Winamp project offers a good suggestion of the possibility's in speech recognition today. Use of natural language dialogue interfaces to standard software is far from perfect. The project did however show that the technical part of voice recognition has come far and will probably be good enough for serious general usage in a couple of years. There are however still the complexity of linguistics, natural language dialogue, to solve in order to use speech control in a natural way. Combinations of words and sentences are close to endless which makes natural dialogue extremely multifaceted. The idea of limiting the field of possible recognition, as in the Winamp project, has however showed that that natural dialogue is possible in very restricted domains.

References

Microsoft Research, *Lifelike Computer Characters: the Persona project at Microsoft Research*, 1996

Nugues Pierre, LTH Lecture notes, 2003

Winamp Homepage: http://www.winamp.com

Microsoft SpeechSDK Homepage: http://www.microsoft.com/speech/download/sdk51/

9 Appendix

A scenario: using the different features of the program

When program starts it outputs (speech output) the following:

Welcome Pierre. This is the Winamp speech control. The jukebox contains, Madonna, Sting, Prince, Red hot Chilipeppers and David Gray. You can choose to list or to play one of these artists. Have a nice try…

The interaction continues with the following dialogue:

Command to system	**Answer from system**	**Action**
Help	*Choose to play a song by an artist or choose to list available songs by artist*	No action
List Madonna	*Loading list, Madonna list <displays (speech) all songs>*	List(Madonna)
Play number two	*Playing*	Play(2)
Stop	*Stop playing*	Stop()
Play next	*Next song*	Play(next)
Play previous	*Previous song*	Play(prev)
Help	*Choose to play next son, previous song, clear list or stop playing*	
Pause	*Pause*	Pause()
Pause	*Pause*	Play()
Stop	*Stop playing*	Stop()
Erase list	*Clear list*	Clearlist()
List artist	*The artists are <displays (speech) all artists>*	No action
Could you find anything by Prince	*Loading list, Prince list <displays all songs>*	List(Prince)
Play	*playing*	Play()
Erase list	*Erasing list*	Clearlist()
I want to hear a song by David Gray	*Playing, playing David Gray*	Play(David Gray)
Stop	*Stop*	Stop
Erase list	*Clearing list*	Clearlist()

A Classification System Applied to Music Reviews

Carl-Emil Lagerstedt
Department of Computer Science
Lund University
dat01cal@ludat.lth.se

Örjan Berglin
Department of Computer Science
Lund University
dat01orb@ludat.lth.se

Abstract

This paper describes a system for classification of music reviews. The system uses a clustering algorithm to build a tree out of a corpus of reviews. Reviews are clustered together based on the similarity of their contents, thereby providing a way to make suggestions of similar artists. Our results show that this approach has potential and should be further explored.

1 Introduction

Staying updated in the world of music is not an easy task. New artists and genres emerges everyday and the most obvious way of learning about new artists is by reading articles and reviews. This can be quite time consuming, and therefore we wanted a way to classify reviews to support the selection of the interesting ones.

This paper describes such a system for classification based on mutual information.

The classification algorithm computes the intersection of two documents and returns a fractional value between 0 and 1, called *document similarity*. The closer the value is to 1, the more the documents resemble each other. A tree is then built based on the document similarities.

2 Purpose

The purpose of this paper is to describe a method for automatically making recommendations, based on the content of reviews. We believe that such a system could be useful in many cases, such as in online shopping. A site could keep a large database of reviews and use them to make suggestions to customers. The benefits of this system is to reduce manual labour.

There is also the possibility that the system will be able to visualize hitherto unknown or unexpected connections between artists, that might not be visible while using traditional keyword-based indexing.

3 Background

Often there is a need to find and show information that is related to a certain topic, for instance, many shopping sites on the Internet have some way of showing recommendations to the customer based on the item currently viewed. This can make it easier for the customer to find interesting objects, and might increase sales.

The methods for producing this information are often primitive, and are often based on the shopping habits/recommendations of previous customers/visitors. Many times this approach is adequate, but the method has obvious drawbacks. Information about previous customers might not be enough or even applicable to this customer. For instance, while browsing crime fiction, a customer is probably not interested in the fact that someone else has bought a cookbook together with the crime novel.

There are also methods that are based on manual labour, where someone has to enter keywords by hand.

This is a method that works quite well, but it can demand a lot of manpower, and due to it's nature is error prone.

We will explore three websites that all have some kind of recommendation system. We do not claim to have exact knowledge about the systems that these sites use. This is just our impression of how the systems work.

3.1 The All Music Guide[1]

The All Music Guide (AMG) is an online resource of artist biographies and record reviews. They have almost 250,000 reviews in their database.

In the biography for an artist, AMG presents a list of similar artists. This list is compiled from data manually entered by AMG's editors and visitors to the site.

3.2 Amazon.com[2]

Amazon.com makes recommendations based on what other buyers of an item has bought. The system seems to be based entirely on the sales statistics, and thus the system does not make intelligent suggestions. Our research shows that this system works quite well, with two exceptions. The first is that one might get only suggestions of the same author. The second is that if an item has not been bought by anyone, no suggestion is made, since there is no sales statistics for that item.

3.3 The Internet Movie Database[3]

The Internet Movie Database describes their recommendation system like this:

"With over 384,000 titles on the IMDb it isn't feasible to handpick Recommendations for every film. That's why we came up with a complex formula to suggest titles that fit along with the selected film and, most importantly, let our trusted user base steer those selections. The formula uses factors such as user votes, genre, title, keywords, and, most importantly, user recommendations themselves to generate an automatic response."

3.4 Background Summary

Even though all the described sites have many reviews, none of the systems makes use of linguistic methods for producing recommendations. In fact, we have failed to find *any* such system.

4 The Music Reviews

Based on the observation that reviews often references other artists, we draw the conclusion that this would be useful in the classification process. For example, in a review of the album *Gold* by *Ryan Adams*, the reviewer mentions that "*…the album is an impressive exploration of territory previously covered by Bob Dylan, Neil Young and other greats before him*". It does not seem far-fetched that a *Ryan Adams* fan might also enjoy *Bob Dylan* and *Neil Young*.

It also seems likely that if artist *A* has received reviews that are similar to those of artist *B*, then their work might also be similar.

The corpus of reviews was collected from several online resources, as well as OCR-ed from print magazines. The corpus consists of a few hundred reviews, selected by us. The reviews cover popular music from the 1960s up until the present date. Therefore our personal preferences may have influenced the selection.

5 Classification Algorithm Overview

We use a clustering algorithm based on intersection. Each document is considered to be a set of words and the intersection of two documents represents how similar they are.

Each document is considered a node. Each node is compared to every other node and the pair with the greatest similarity is selected out. A new node is created with the two selected nodes as its children and reinserted into the set of all nodes, containing all the words from the two nodes.

This is repeated until there is only one node left. In the resulting tree, nodes that are close should have similar content.

[1] http://www.allmusic.com
[2] http://www.amazon.com
[3] http://www.imdb.com

5.1 Document Similarity

The union of two documents is the base for the *document similarity*.

A fractional number between 0 and 1 is returned by the following equation:

$S = |D_1 \cup D_2| / (|D_1| + |D_2|)$,

where D_1 and D_2 are the documents and S is the document similarity. A high document similarity indicates that the two documents have similar content. The document similarity is used only on the lowest level of the tree. When compound nodes are compared, the set of words in each node's children are computed recursively.

5.2 Algorithm Efficiency

Since, in every iteration, each node has to be compared to every other node, the algorithm has a time complexity of $O(n^2)$, where n is the number of nodes. Some of the computation could be eliminated by using dynamic programming. Even though the algorithm is slow, it is not a major impediment, since the tree needs to be built only once. The only time the tree must be rebuilt is when additional reviews are added to the database.

6 Prototype Implementation

Our prototype is coded entirely in Java and makes heavy use of hash tables to store document content. We use a stop list to remove common words that provides little or no information. This reduces the number of comparisons in the algorithm. As a consequence, the algorithm runs faster, and also more accurate. The reviews are stored as plain text files. Output is visualized using *GraphViz* from AT&T Labs-Research[4]. GraphViz produces a visual graph that can be zoomed and panned to study the results. The running time for building a tree with 240 reviews is about 60 minutes on a 2.0 GHz Pentium 4.

7 Results

Our initial idea, that we could group similar reviews together, proved to work pretty well. Based on observations of our implementation, we estimate that about 60-70% of the reviews are clustered correctly.

We saw that one important factor was the quality of the corpus. If a review is too short, it doesn't contain very much factual information, and is therefore not very usable in the clustering effort.

One other objective would be to produce a more balanced tree. This way we could easily cut the tree at an arbitrary level, to produce clusters. With the current implementation, the tree can become very unbalanced.

Similar artists do get clustered together. Efficiency is, however, hard to measure, since different people might have different views on how the artists should be grouped. An example of a succesful (in our opinion) grouping, is that of *Ryan Adams* and *Bob Dylan* getting grouped together. An example of an obviuos grouping is that *Bob Dylan* reviews gets grouped with other *Bob Dylan* reviews.

However, some reviews get clustered in at a completly wrong place in the tree. For example, *Velvet Underground* has, in our tests, been clustered with *Abba* as its closest neighbour, which, in our opinion, feels totally wrong. There should be better matches for both these bands. We believe that this bad matching has two reasons. First, in our implementation, all documents are eventually put in the tree, regardless of whether they have any similarity to an other document. Second, we have a pretty limited corpus. A solution to the first problem is to have a threshold value for the document similarity. This would eliminate the insertion of irrelevant nodes.

8 Conclusion

We are satisfied with our results, even though they might not seem very impressive. Our proposition seems to hold, that is, an automated clustering of reviews is a useful idea.

More work needs to be done, but we are confident that, given enough time and effort, this could also turn out to be a useful application.

4 http://www.research.att.com/sw/tools/graphviz/

9 Future Directions

We would like to implement the vector space model, to compare the efficiency and the results of the algorithms.

When a review has very little in common with other review, there is no point in inserting it in the tree. As previously mentioned, a threshold value for the document similarity could be used to remove those reviews.

The use of an inverted index (where rare words are weighted up) should be explored.

The ability to detect bold and italicised words might provide additional clues as to what words in the review are important.

A name extraction feature would further enhance the similarity values, since artist names commonly are used as references when writing a review.

To reduce the running time of the algorithm, dynamic programming should be utilized to reduce the number of comparisons that need to be made.

To speed-up the application, distributed data processing could be used. It would be interesting to test the algorithm on a distributed system, using a large corpus, such as The All Music Guide.

Finite state clause segmentation

Anders Berglund
Lund Technical Highschool
`d98ab@efd.lth.se`

Abstract

This report describes an attempt to perform clause segmentation in Swedish using a method published by Eva Ejerhed in (András, 1999) pp. 140-151.

1 Credits

The method I used is basically the method developed and published by Eva Ejerhed, the grammar I use is hers entirely and this very project wouldn't have been possible without her paper.

2 Introduction

Ejerhed's idea was to use a finite state automaton that triggers on certain patterns of word tags and inserts clause boundaries. The automaton is easy to specify and easy to implement, the algorithm isn't too computationally intensive and the results good, Ejerhed reports a degree of correctness of 96% with the method applied to manually tagged text.

3 Basic Overview

The finite state automaton in my implementation takes a tagged text as input, metaclassifies the tags (the automaton has a metatag "FIN" meaning a finite verb, all words classified as verbs being in a finite form gets metaclassified as "FIN") as a first step, then runs the rules given in the automaton using simple pattern matching (the automaton rules can be found in the appendix).

The metatags are quite few, around 10, and they do not overlap. As they are that few, a simple solution is giving each an integer value of its own and using those integers in the computations, making it possible to avoid string compares in the second step.

In the second step my pattern matcher simply runs *"are the next n tags (tag x tag y tag z)? then tag c after tag x"* in a long loop with different patterns of different length, advancing the automaton one step after each iteration of the loop.

4 Results

Eva Ejerhed remarks in her paper that there are many open questions concerning the definition of the clause units to have as targets for clause segmentation. One aspect she notes is whether a clause should have *at most* or *exactly* one finite verb per clause.

Eva Ejerhed chooses to have at most on finite verb per clause, and this leads to a clause segmentation that looks sometimes looks odd. An example of sentence from the *aa01* text from the SUC1A-corpus tagged with clause delimiters given by the state machine (quite thoroughly debugged, it *does* follow the rules):

```
<C1A> Don Kerr , en av de
politiska tänkarna pá det ansedda
analysinstitutet IISS , <C1C> är
inte páfallande optimistisk
<C2Bg> när han talar om saken .
```

The `<C1C>`-clause segmentation is strange

(<C1C> means a cl ause segmentation marker that was produced by rule 1C). In my book the entire sentence is a single clause (at least the part before the <C2Bg>-tag (the entire part before the <C2Bg> constitutes the subject of the phrase)), the automaton "overrecognizes" when it encounters something that fits rule 1C. A solution might be having a "flag" in the automaton that keeps track of whether the preceding stretch of words starting with the last DL MID-tag contained any verb, if not, then rule 1C shouldn't be applied. This flag would be set to *false* at every encounter of a delimiter and set to true when encountering a verb.

Such a change would mean a change of focus to having *exactly* one finite verb per clause. Eva Ejerhed reports very few *overrecognized* clauses, using a manually annotated corpus (which the SUC1A-corpus is) she finds *zero* overrecognized clauses. The Swedish construct above, what in (at least) Swedish grammatic terminology for German is called an *apposition* [1], is quite common, and I think it has been overrecognized in this case.

This is directly at odds with Ejerhed's results (*she*: zero overrecognizations, *I*: common construct overrecognized), but I'm reluctant to take a stand and I choose to leave the question of who is right as an exercise to the reader.

5 Conclusion

The complaint above apart, the method works very nicely for most cases, is easy to implement (and debug!) and not very computationally expensive. The performance attained with such simple functionality is quite impressive.

References

András Kornai (Ed.) 1999. *Extended finite state models of language*. Cambridge University Press.

[1]An example of an apposition: *Berlin, the German capital, is big.* The string *the German capital* is an attribute to the preceding noun *Berlin*, an attribute without a verb written between commas. As there is no verb it is not a clause of its own, and it should be treated as were it an adjective on the top-level.

Appendix: The automaton rules from Ejerhed (András, 1999) pp. 140-151

Clause segmentation rules

```
1 PUNCTUCATION
1a1) <h>XX -> <h><c>XX
1a2) <p>XX -> <p><c>XX
1b) DL-MAD XX -> DL-MAD <c> XX,
where XX is not end tag
1c) DL-MID FIN -> DL-MID <c> FIN
1d) DL-MID XX FIN -> DL-MID <c>
XX FIN, where XX=PN, NN, PM or AB

2 COMPLEMENTIZERS
2as) XX KN SN -> XX <c> KN SN
2ag) XX SN -> XX <c> SN
2bs) XX KN HX -> XX <c> KN HX
2bg) XX HX -> XX <c> HX

3 KN + FINITE VERB
3s) XX KN FIN -> <c> XX KN FIN,
where XX is a closed class of
finite forms of the verbs be, go,
stand, sit
3g) XX KN FIN -> XX <c> KN FIN

4 KN + XX + FINITE VERB, where
XX=PN, NN, PM or AB
4s) YY KN XX FIN -> <c> YY KN XX
FIN, if YY=XX
4g) YY KN XX FIN -> YY <c> KN XX
FIN, if YY!=XX

5 SEQUENCES OF FINITE VERBS
5a) CASE: 0 WORDS BETWEEN FINITE
VERBS
FIN FIN -> FIN <c> FIN
5b) CASE: 1 WORD BETWEEN FINITE
VERBS
FIN XX FIN -> FIN XX <c> FIN
5c) CASE: 2 WORDS BETWEEN FINITE
VERBS
5cs) FIN YY XX FIN -> FIN YY <c>
XX FIN, where XX=PN, NN or PM
5cg) FIn YY XX FIN -> FIN YY XX
<c> FIN
```

ABBREVIATIONS
<h> = head
<p> = paragraph
</h> = end head
</p> = end paragraph
DL MAD = major delmiter (.?!)
DL MID minor delimiter (,-:)
FIN = finite verb; VB PRS AKT, VB PRS SFO, AB PRT AKT, VB PRT SFO, VB SUP AKT, VB SUP SFO, VB IMP AKT
PN = PN ... SUB, PN ... SUB/OBJ (subject forms of pronouns)
NN = NN ... NOM (nouns)
PM = PM NOM (proper nouns)
AB = AB, AB POS, AB KOM, AB SUV (adverbs)
KN = conjunction
SN = subjunction
HX = HA, HD ..., HP ..., HS ... (Wh: adverbs, determiners, pronouns, possesives)

Sats-segmentering för svenska

Andreas Brandt
University of Lund
dat01anb@ludat.lth.se

Abstract

In this report I describe a method for segmenting POS-tagged swedish discourse into clauses.

I have implemented this method for two different tag-sets, and evaluated the results.

1 Introduktion

Att dela upp en text i satser är inte något som man ofta har som slutmål i en applikation, men ofta har man stor användning av det då man behöver analysera semantiken i en text. I stället för att analysera meningar, som ofta har flera semantiska betydelser, är det lättare att analysera segment som var för sig har en innebörd. Användningsområdena är främst andra lingvistiska applikationer, så som t.ex. talsyntes, röstigenkänning, textanalys, maskinöversättning och kunskapsinsamling. Det är tänkt att Carsim[1] ska använda denna metod för att extrahera information ur text, och konvertera det till scener.

Styrkan hos den här metoden är att den är robust eftersom den inte kräver någon generering av parse-träd och liknande, utan bara en genomgång av texten.

2 Bakgrund

En sats definieras här utifrån hur vi uppfattar en uppläst text. I studier utförda av (E. Strangert, 1993) använder man akustisk information, i form av tysta intervall och tonfall, och perceptuellt uppfattade pauser för att placera sats-avskiljare. Det visar sig att alla dessas, samt den algoritmiska metodens(Ejerhed, 1990), avskiljare inte är oberoende. Utan tvärtom, den algoritmiska metodens avskiljare sammanfaller nästan alltid med någon av de andra. Detta visar att metoden verkar fungera.

[1]http://www.df.lth.se/ richardj/carsim/

3 Metod

Indata till sats-segmenteraren är en POS-taggad text, med taggar enligt SUC[2].

Texten gås igenom och varje bigram, trigram eller 4-gram, kontrolleras mot en uppsättning av regler(Ejerehed, 1996)(se Appendix). Reglerna talar om var en sats-avgränsare ska sättas in.

4 Implementation

Implementeringen skedde i Java, med dess paket för reguljära uttryck[3]. Implementationen stöder både SUC's och Granska's[4] tagg-uppsättning.

4.1 SUC

Det finns två olika format på SUC, jag valde SUC1A. Det kan se ut så här:

```
("<Vi>" <86>
        (PN UTR PLU DEF SUB "vi"))
("<sitter>"     <87>
        (VB PRS AKT "sitta"))
("<här>"        <88>
        (AB "här"))
("<i>"  <89>
        (PP "i"))
("<solen>"      <90>
        (NN UTR SIN DEF NOM "sol"))
("<tills>"      <91>
        (SN "tills"))
("<det>"        <92>
        (PN NEU SIN DEF SUB/OBJ "det"))
("<är>" <93>
```

[2]Stockholm Umeå Corpus
[3]java.util.regex.*
[4]En POS-taggare av Johan Carlberger

```
        (VB PRS AKT "vara"))
("<dags>"       <94>
        (NN UTR - - - "dags"))
("<.>"  <95>
        (DL MAD "."))
```

Vi plockar ut varje symbol(token), tagg och lemma och stoppar in i varsin array. Resultatet blir att symbol[n] har tagg[n] och lemma[n]. Därefter blir det mycket enkelt att kontrollera bi-gram, tri-gram och 4-gram. Vi har även en array av samma storlek som de övriga, med sats-avgränsare. Finner vi att det enligt reglerna ska vara en sats-avgränsare på position n, så sätter c[n] = sant. Därefter har man all nödvändig information.

Exempel på utdata:

```
<c> Den kallas allmänt stenbit , </c>
<c> men det är bara hannarna </c>
<c> som heter så </c>
```

4.2 Granska

Jag har även implenterat en segmenterare för Granska. Granska har en annan uppsättning av taggar, men så gott som alla nödvändiga kategorier finns med. Det saknas taggar för paragrafer och överskrifter, men dessa spelar en underordnad roll. En text taggad med Granska kan se ut så här:

```
Vi      pn.utr.plu.def.sub      vi
sitter  vb.prs.akt      sitta
här     ab      här
i       pp      i
solen   nn.utr.sin.def.nom      sol
tills   sn      tills
det     pn.neu.sin.def.sub/obj  det
är      vb.prs.akt.kop  vara
dags    nn.utr  dags
.       mad     .
```

5 Utvärdering

Sats-segmenteringen fungerar förvånansvärt bra, och många av feltaggningarna tycks kunna gå att kringå med exempelvis specialfall. En del fel beror också på hur texten är POS-taggad, inte bara fel-taggning, utan också vilken policy som använts.

I (Ejerehed, 1996) uppges att metoden ger 96% korekt taggning på SUC. Detta ligger nog ganska nära sanningen, jag har kontrollerat ett textstycke med 98 funna satser. Jag hittade tre fall som taggats fel, vilket ger ungefär samma siffra. Att finna fel är inte helt trivialt, då man oftast får göra en mer eller mindre subjektiv bedömning.

5.1 Feltaggning

Enligt specialfallet i regel 4(se Appendix) ska en brytning ske här:

```
<c>
Vi      (PN UTR PLU DEF SUB)
träffas         (VB PRS SFO)
</c>

<c>
där     (AB)
och     (KN)
sen     (AB)
tar     (VB PRS AKT)
vi      (PN UTR PLU DEF SUB)
oss     (PN UTR PLU DEF OBJ)
hit     (AB)
till    (PP)
.
.
.
```

Här borde en brytning skett:

```
<c>
Om      (SN)
vi      (PN UTR PLU DEF SUB)
inte    (AB)
ses     (VB INF SFO)
så      (AB)        <---
ses     (VB INF SFO)
vi      (PN UTR PLU DEF SUB)
om      (PP)
tre     (RG NOM)
månader         (NN UTR PLU IND NOM)
här     (AB)
.       (DL MAD)
</c>
```

5.2 Gränsfall

Ett ganska vanligt förekommande fall av ett tveksamt fel, är när hjälpverbet “ha” förekommer. Här kan man tycka att ingen satsbrytningen skulle ha skett.

```
<c> Jag har </c>
<c> fått pengar </c>
```

6 Förslag

Eftersom SUC har en reltaivt fattig uppsättning av taggar, kan man säkerligen hitta fler regler med en större uppsättning. Exempelvis fattas taggning för olika former av hjälpverb i SUC. Med t.ex. Granskas mer nyansrika taggar är det kanske möjligt att skriva regler för vissa fall som inte kan fångas upp annars.

7 Slutsats

Sats-segmentering med sats-avgränsare istället för metoder som använder sig av nästlade satser(sats, bi-sats), är en metod som är både enkel att implementera och effektiv att använda.

Det verkar även finnas bra möjligheter att förbättra metoden om man har nyansrikare taggar, som i Granska.

Acknowledgements

Tack till Richard Johansson för hjälp och tips.

References

D. Huber E. Strangert, E. Ejerhed. 1993. Clause structure and prosodic segmentation. *Papers from the Seventh Swedish Phonetics Conference*.

E. Ejerehed. 1996. Finite state segmentation of discourse into clauses. *Extended Finite State Models of Language*.

E. Ejerhed. 1990. A swedish clause grammar and its implementation. *The 7th Nordic Conference on Computational Linguistics*.

Appendix

A Stycket som utvärderades

```
<c> Jag vet inte . </c>
<c> Du får sova hos mig , </c>
<c> sa Lena så tyst </c>
<c> att bara Torvald hörde det . </c>
<c> Men det kan inte bli som förra
gången . </c>
<c> Jag lovade Matt . </c>
<c> Jag förstår , </c>
<c> sa Torvald . </c>
<c> Min ryggsäck står där inne under
flipperspelet , </c>
<c> sa Gunnar . </c>
<c> Hur full jag än är </c>
<c> måste jag och Rosita i väg .</c>
<c> Det ordnar sig , </c>
<c> sa Torvald . </c>
<c> När kommer Rosita hit . </c>
<c> Jag vet inte , </c>
<c> sa Gunnar . </c>
<c> Men hon kommer . </c>
<c> Hon måste ju ha </c>
<c> förstått att snuten höll mig kvar .</c>
--<c> Jag har </c>
<c> suttit hos snuten i natt , </c>
<c> sa han sen förklarande till alla </c>
--<c> som eventuellt inte hade </c>
<c> hört det . </c>
<c> I det ögonblicket stannade Bill Marshall
sin Morgan utanför Monaco . </c>
<c> Torvald , </c>
<c> sa han . </c>
<c> Kan vi åka en sväng . </c>
<c> Jag måste snacka med dig . </c>
<c> Torvald visste inte mycket om bilar men
en grön Morgan är en grön Morgan </c>
<c> och ser ut som en bil . </c>
<c> Torvald reste sig . </c>
<c> Vi ses på Gaudeamus , </c>
<c> sa han . </c>
<c> Vid lunch . </c>
<c> Om jag inte kommer så </c>
<c> ta mina pengar , Gunnar , </c>
<c> och gör rätt för oss </c>
<c> och bjud på lunch . </c>
<c> Men jag kommer . </c>
<c> Annars är jag på Select
klockan fyra . </c>
<c> Sen fick han en idé . </c>
<c> Han gick fram och </c>
<c> snackade med Bill Marshall </c>
<c> och kom sen tillbaka . </c>
<c> Han sa till Gunnar : </c>
<c> Bill kör dig och Rosita Zoraffie
till stationen . </c>
<c> Han hämtar er här halv sex . </c>
<c> Om vi inte ses , Gunnar , </c>
<c> vilket vi gör , </c>
<c> men om vi inte ses så </c>
<c> skriver jag till dig . </c>
<c> Om några månader är det jobb i
```

```
Schweiz på tretusen
meters höjd över Sion . </c>
<c> Vi anmäler oss dit . </c>
--<c> Vi träffas </c>
<c> där och sen tar vi oss hit till
Monaco igen för en månadsfylla </c>
<c> och sen sticker vi hem . </c>
<c> Hälsa nu alla . </c>
<c> Rör inga kalabriska flickor . </c>
<c> Krama om Peppino och
den döva killen . </c>
<c> Jag avundas dig lite . </c>
<c> Behåll pengar </c>
<c> så ni åtminstone kommer lite
söder om Neapel . </c>
<c> Det som är kvar </c>
<c> kan du ge till Bibi . </c>
<c> Okey , </c>
<c> sa Gunnar . </c>
<c> Om vi inte ses så ses vi om
tre månader här . </c>
<c> Jag ska forska i schweizjobbet . </c>
<c> Torvald klev in i Bill Marshalls </c>
<c> Morgan och Bill sa : </c>
<c> Vi sticker till ett ställe </c>
<c> där vi får vara i fred . </c>
<c> Vad tror du om kallt vitt vin , </c>
<c> sa Torvald . </c>
<c> Det tror jag på , </c>
<c> sa Bill . </c>
<c> Kör till Saint-Sulpice , </c>
<c> sa Torvald . </c>
<c> Jag ska visa dig ett nytt ställe </c>
<c> där vi får vara i fred . </c>
<c> Dig har jag </c>
<c> önskat träffa </c>
<c> sen livet upphörde för nån dag sen .</c>
<c> I lördags . </c>
<c> De for iväg </c>
<c> och Bill parkerade bilen i närheten
av den rostfria baren </c>
<c> och sen gick de in där . </c>
<c> De möttes av en glad och pigg Jerry </c>
<c> som omedelbart presenterade sig
för Bill Marshall </c>
<c> och hämtade en flaska av sitt vin .</c>
<c> Han hämtade två glas och en askkopp .</c>
<c> Han slog i glasen </c>
<c> och sen lät han dem vara i fred . </c>
<c> Har du letat efter mig , </c>
<c> sa Bill . </c>
<c> Ja , </c>
<c> sa Torvald . </c>
```

B

B.1 Regler(Ejerehed, 1996)

```
1 PUNCTUATION
1a
<h> XX => <h> <c> XX
<p> XX => <p> <c> XX

1b
DL MAD XX => DL MAD <c> XX,
where XX is not end tag

1c
DL MID FIN => DL MID <c> FIN
1d
DL MID XX FIN => DL MID <c> XX FIN,
where XX= PN, NN, PM or AB


2 COMPLEMENTIZERS

2a
special:
XX KN SN => XX <c> KN SN
general:
XX SN => XX <c> SN

2b
special:
XX KN HX => XX <c> KN HX
XX HX HX => XX <c> HX HX
general:
XX HX => XX <c> HX


3 KN+FINITE VERB

special:
XX KN FIN => <c> XX KN FIN,
where XX is a closed class of
finite forms of the verbs
‘‘vara’’, ‘‘gå’’, ‘‘stå’’, ’’sitta’’
general:
XX KN FIN => XX <c> KN FIN$


4 KN+XX+FINITE VERB,
where XX=PN, NN, PM or AB

special:
YY KN XX FIN => <c> YY KN XX FIN,
 if YY=XX
general:
YY KN XX FIN => YY <c> KN XX FIN,
 if YY<=XX


5 SEQUENCES OF FINITE VERBS

5a CASE: 0 WORDS BETWEEN FINITE VERBS
FIN FIN => FIN <c> FIN

5b CASE: 1 WORD BETWEEN FINITE VERBS
FIN XX FIN => FIN XX <c> FIN
```

```
5c CASE: 2 WORDS BETWEEN FINITE VERBS
special:
FIN YY XX FIN => FIN YY <c> XX FIN,
where XX=PN, NN, or PM
general:
FIN YY XX FIN => FIN YY XX <c> FIN
```

B.2 Taggar(Ejerehed, 1996)

```
<h>
 head

<p>
 paragraph

<<<<kk09>>>>
 block

</h>
 end head

</p>
 end paragraph

<<<</kk09>>>>
 end block

DL MAD
 major delimiter ( . ? ! )

DL MID
 minor delimiter ( ,  : )

FIN
 VB PRS AKT, VB PRS SFO,
 VB PRT AKT, VB PRT SFO,
 VB SUP AKT, VB SUP SFO,
 VB IMP AKT

PN
 PN ... SUB, PN ... SUB/OBJ
 (subject forms of pronouns)

NN
 NN ... NOM

PM
 PM NOM

AB
 AB, AB POS, AB KOM, AB SUV
 (adverbs)

KN
 conjunction

SN
 subjunction

HX
 HA, HD..., HP..., HS...
 (Wh: adverbs, determiners,
  pronouns, possessives)
```

Name Extraction in Car Accident Reports for Swedish

Lisa Persson
Department of Computer Science
Lund University
nossrepasil@hotmail.com

Magnus Danielsson
Department of Computer Science
Lund University
mangedl@hotmail.com

Abstract

This report is part of the work included in a course of computer linguistics. Our task was to implement a program in Java to tag a corpus of descriptions of car accidents in Swedish. The words we should tag were geographical places, primarily names. To do this we first tag words that should not be included in the final result, just to be able to exclude them. Such things are brands of cars, counties, countries and proper names. Later we go on to tag locations, streets, roads, highways, cities and squares.

To evaluate the program we calculate recall and precision values, comparing a small manually tagged corpus with the same corpus tagged by the program.

1 Inledning

Syftet med den här rapporten är att beskriva en metod att märka upp olika typer av namngivna platser i en korpus. Den korpus som används består av en samling nyhetsnotiser om bilolyckor. För uppmärkningen används ENAMEX-taggar[1]. De typer av platser som taggas är: orter, platser, torg, gator och vägar.

Programmet är tänkt att användas inom projektet CarSim[2], ett text-till-scenomvandlingsprogram för trafikolycksrapporter ("text-to-scene conversion system for traffic accident reports"). CarSim används för att läsa in en text på svenska som beskriver en bilolycka på naturligt språk och sedan extrahera så pass mycket information om händelseförloppet att det går att göra en grafisk simulering.

2 Taggningsregler

Programmet skall tagga sex olika typer av platser. Nedan följer en beskrivning av dessa taggar och deras definitioner.

2.1 CITY

CITY-taggar används för uppmärkning av namngivna städer, byar och stadsdelar. Exempel på fraser som skall märkas med CITY-taggar är ”Stockholm”, ”Västra Frölunda” och ”Limhamn”. För att få märkas upp med en CITY-tagg måste ortsnamnet stå för sig själv och inte vara en del av ett ord, t.ex. taggas inte ”Stockholmsområdet”. Ortsnamnet får inte heller vara en del av en annan tagg. I frasen ”Malmö Allmänna sjukhus” ingår ortsnamnet ”Malmö” men denna fras definieras som en plats och skall märkas upp med en LOCATION-tagg. Däremot taggas ortsnamn med en CITY-tagg i alla övriga fall, även om det inte är orten i sig som avses, t.ex.:

```
”... hon spelade i <ENAMEX
TYPE=”CITY”>Umeå</ENAMEX>
I.K.”
```

2.2 LOCATION

[1] Andrei Mikheev, Marc Moens and Claire Grover. 1999.
[2] Per Andersson, 2003.

LOCATION-taggar används för uppmärkning av namngivna platser. Undantaget är torg som taggas med SQUARE-taggar. Exempel på platser som skall märkas med LOCATION-taggar är: “Eneby kyrka”, ”Universitetssjukhuset Mas” och ”Globen”. Grundregeln för att en fras skall få märkas med en LOCATION-tagg är att alla ingående ord i frasen är en del av platsens egennamn. De ord som endast beskriver vilken typ av plats det rör sig om skall ej ingå i taggen, några exempel:

```
”... till
universitetssjukhuset
<ENAMEX
TYPE=”LOCATION”>MAS</ENAMEX>
...”

”... till <ENAMEX
TYPE=”LOCATION”>Universitets
sjukhuset MAS</ENAMEX> ...”

”... vid domkyrkan i ...”

”... vid <ENAMEX
TYPE=”LOCATION”>Domkyrkan</E
NAMEX> i ...”

”... förbi <ENAMEX
TYPE=”LOCATION”>Sörunda
ridskola</ENAMEX> ...”
```

I det första och tredje exemplet är “universitetssjukhuset” resp. ”domkyrkan” en beskrivning av vilken typ av plats som avses och skall därmed ej taggas. I det andra och fjärde exemplet är ”Universitetssjukhuset” resp. ”Domkyrkan” en del av platsernas namn och skall därför taggas. I dessa fall kan man sluta sig till detta beroende på om orden börjar på stor eller liten bokstav. I det sista exemplet börjar ”ridskola” på liten bokstav men av sammanhanget kan man sluta sig till att ordet är en del av ett platsnamn.

2.3 SQUARE

SQUARE-taggar används för uppmärkning av namngivna torg. Samma regler som för LOCATION-taggar om att endast de ord som är en del av egennamnet skall ingå i taggen gäller. Exempel på torg: ”Gustav Adolfs torg”, ”Stortorget” och ”Olof Palmes plats”.

2.4 STREET

STREET-taggar används för att märka upp namngivna stadsgator. Exempel på gatunamn som skall märkas med STREET-taggar är: ”von Rosens väg” och ”Storgatan”.

2.5 ROAD

ROAD-taggar används för att märka upp namngivna landsvägar, riksvägar, länsvägar och större genomfartsleder. Exempel på vägnamn som skall märkas med ROAD-taggar är: ”länsväg 225” och ”riksväg 67”.

2.6 HIGHWAY

HIGHWAY-taggar används för att märka upp namngivna motorvägar och europavägar. Exempel på vägnamn som skall märkas med HIGHWAY-taggar är: ”Europaväg 4” och ”E 22”.

3 Metoder

Vid taggningen av korpusen har en mängd metoder använts. Dels har databaser med namngivna platser använts och dels ett antal regler. Den kanske mest grundläggande regeln för fraser som skall märkas med CITY-, LOCATION- eller SQUARE-taggarna är att de nästan alltid inleds med ett ord med en inledande stor bokstav. Denna iaktagelse underlättar taggningsarbetet – en majoritet av alla korpusens ord inleds med liten bokstav och kan därför uteslutas. Men detta medför även svårigheter, samtliga meningar inleds av ord som börjar på stor bokstav. Det finns även en mängd andra typer av ord som inleds med stor bokstav, som egennamn, länder, landskap, sjöar, bilmodeller m.fl. En lösning på detta problem som visade sig vara fruktbart var införandet av det vi valt att kalla hjälptaggar.

3.1 Hjälptaggar

Det finns fraser i korpusen som taggas men där taggningen inte syns i slutresultatet. De taggar som används till detta kallar vi med ett samlingsnamn för hjälptaggar. Hjälptaggarna är precis som övriga taggar ENAMEX-taggar[3]. Tre olika hjälptaggar används, REMOVE, AREA och NAME.

Det viktigaste skälet att använda hjälptaggar är att förhindra att de fraser som märks med hjälptaggar inte blir felmärkta med någon av de ”riktiga” taggarna. Precis som för taggarna CITY, LOCATION och SQUARE inleds de fraser som taggas med hjälptaggar oftast av ord som inleds med stor bokstav. Det är dessa taggar som riskerar att märkas fel utan hjälptaggarna.

Hjälptaggarna används under exekveringen på samma sätt som de andra taggarna men raderas från resultattexten i slutet av exekveringen.

Det finns flera skäl att låta hjälptaggarna vara av ENAMEX-typ samt ha olika namn trots att de skall tas bort. Framförallt med NAME-taggarna, som taggar personnamn, uppnåddes bra resultat. Genom att de följer ENAMEX-konventionen för personnamn kan man därför låta dessa taggar vara kvar i texten om man i framtiden skulle ha behov för detta.

Internt, för programmet, är det en stor hjälp att de olika hjälptaggarna har olika namn. Regler för både hjälptaggar och övriga taggar utnyttjar ibland de intilliggande taggarna. Det inträffar även att taggtypen behöver ändras från t.ex. en hjälptagg till en annan tagg genom att taggen står i en viss kontext i korpusen. Här följer de regler som gäller för hjälptaggarna:

3.1.1 NAME

Hjälptaggen NAME används vid uppmärkning av personnamn. Endast namn på personer och inte t.ex. företags- eller organisationsnamn taggas med NAME-taggar. Även här gäller att det endast är själva namnet som taggas och inte syftningar på personen. Dock taggas smeknamn, t.ex. ”Jan "Blondie" Hammarlöf” med NAME-taggar. Titlar ingår i NAME taggen om de börjar på stor bokstav, t.ex. ”Drottning Silvia”.

3.1.2 AREA

Hjälptaggen AREA används vid uppmärkning av fysiskt utspridda landområden samt ”ospecifika” platser. T.ex. taggas länder, sjöar, större öar, län och landskap med AREA-taggar. Till ”ospecifika” platser räknas t.ex. ”Sydsverige”, ”Boråsområdet” och ”Norrtäljetrakten”.

3.1.3 REMOVE

Hjälptaggen REMOVE används för en mängd olika kategorier av ord. Gemensamt för dessa är att de inleds med stor bokstav och inte används för uppmärkning av andra typer av taggar genom kontextregler.

En del i bilolycksartiklar vanligt förekommande förkortningarna taggas med REMOVE-taggar. De är inte så många men vanligt förekommande, t.ex. ”SOS”, ”SJ”, ”TT” m.fl. En del ändelseregler används också, exempelvis taggas ord med ändelserna ”polisen”, ”bladet”, ”verket”, ”borna” m.fl. med REMOVE-taggar. Exempel är: ”Jönköpingsborna”, ”Vägverket”, ”Sportbladet” och ”Kalmarpolisen”. Även fraser som inleds med ”Radio” eller ”Nyhetsbyrån” taggas, t.ex. ”Radio Gävleborg” och ”nyhetsbyrån NTB”.

En databas på i Sverige vanligt förekommande bil-/motorcykel- märken/modeller används för att tagga bilar och motorcyklar. I en korpus som handlar om bilolyckor är det naturligtvis vanligt med bilar.

3.2 Stor bokstav

En grundförutsättning för övriga regler är att frasen som skall taggas inleds med stor bokstav. Detta är underförstått i den fortsatta texten vid beskrivning av taggningsregler. Enda generella undantaget är för ROAD-taggar där fraserna ofta börjar på liten bokstav, t.ex. i ”länsväg 209” och ”väg 108”. I den korpus vi använde vid uttestandet av programmet hittades endast ett exempel, för övriga taggar, på att en taggad fras inleddes med liten bokstav. Vi kunde tagga detta undantag, ”von Rosens väg”, korrekt genom att ”von” fanns i vår databas över Sveriges 1000 vanligaste efternamn[4]. Tillsammans med ”af” var detta den enda posten i databaserna som inleddes med liten bokstav.

Även hjälptaggarna inleds med stor bokstav; ett viktigt skäl för införandet av dessa var att utesluta dessa fraser som inleds med stor bokstav och som ej skulle taggas. Ett problem som återstår är alla ord som inleder meningar. Det kan

[3] Andrei Mikheev, Marc Moens and Claire Grover. 1999.

[4] Robert Larsson.

tyckas vara ett stort problem att avgöra om dessa ord skall taggas eller inte. Lösningen på problemet var att undvika det – i de flesta fall skall ord i början av meningar inte taggas. Vi gjorde iaktagelsen att bara i undantagsfall inleds en mening av ett ortsnamn, platsnamn eller vägnamn och denna defensiva strategi gav ett överraskande bra resultat.

Ett par undantag finns dock. I databaserna finns ”säkra” namn som inte kan förväxlas med andra ord. Dock fick en ort som ”Bara” uteslutas från ortsnamnsdatabasen då detta är ett ord som ibland inleder meningar.

3.3 Databaser

Programmet använder databaser med sammanlagt ca. 2500 poster. Med databaser avses här listor med namngivna entiteter. Databaserna är indelade i kategorier efter vilken typ av tagg de används till. Programmet använder följande databaser:

- 896 förnamn
- 760 efternamn
- 148 bil-/motorcykel- märken/modeller[5]
- 6 förkortningar
- 433 svenska ortsnamn
- 191 länder[6]
- 9 svenska landskap
- 4 svenska sjöar

Under utvecklingen av programmet krymte de ursprungliga databaserna för förnamn, efternamn och svenska ortsnamn. Ursprunglingen användes listor med Sveriges 1000 vanligaste för- resp. efternamn[7] samt en lista med 1575 svenska ortsnamn[8]. Reduceringen av databaserna kunde ske tack vare användningen av ändelseregler.

3.4 Ändelseregler

Det första som gjordes i programmet i programutvecklingens inledningsskede var att tagga ortsnamn med CITY-taggar med hjälp av en databas. I detta skede var resultatet ej tillfredställande. I vår testkorpus fanns ett antal mindre orter som ej fanns med i databasen och därmed inte blev taggade. Iaktagelsen att många ortsnamn hade gemensamma ändelser gjordes, t.ex. är: ”borg”, ”ryd”, ”hamn”, ”holm” och ”löv” vanliga ortsnamnsändelser. En regel för att märka ord som började på stor bokstav och avslutades med någon av dessa ändelser infördes. Resultatet blev att fler ortsnamn märktes korrekt, men även att ett stort antal ord som ej var ortsnamn felmärktes med CITY-taggar. Det konstaterades att en stor andel av de ord som feltaggades var i form av personnamn. Vi insåg därför att det fanns ett behov att även tagga personnamn i uteslutningssyfte.

I programmet implementeras ändelseregler för person- och ortsnamn genom 161 ändelser för ortsnamn och fem ändelser för efternamn. Metoden för att få fram vilka ändelser som var lämpliga och vilka som ej var lämpliga var ”trial and error”-baserad. En utprovning skedde, ändelse för ändelse, på testkorpusen och de ändelser som medförde feltaggningar lades ej in i listan.

För SQUARE, STREET, ROAD och LOCATION implementeras ändelseregler genom att fraser som avslutas med vissa nyckelord taggas. Några exempel på detta är: ”torg” eller ”plats” för SQUARE, ”gata” eller ”väg” för STREET, ”led” för ROAD och ”sjukhus”, ”kyrka”, ”hamn”, ”skola” m.fl. för LOCATION. Fraserna taggas även om ändeleorden är i bestämd artikel. Mest används ordändelser för LOCATION, det finns 26 olika ordändelser för LOCATION. Ändelserna kan vara fristående ord eller avsluta ett ord, t.ex. taggas både ”Danderyds sjukhus” och ”Drottning Silvias barnsjukhus” som LOCATION. I det sista exemplet andvänds ordutvidgning för att hitta hela frasen.

Även för hjälptaggarna REMOVE och AREA används ändelseregler. För REMOVE används t.ex. ”borna”, ”tidningen”, ”verket” m.fl. För AREA används ”området”, ”sverige”, ”trakten”, ”län” och ”land” för ändelseregler. En majoritet av Sveriges landskap, landsdelar (Götaland, Svealand och Norrland) samt några länder kunde taggas med ändelsen ”land”.

[5] Susning.nu 2003, Teknikens Värld. 2002.
[6] Wikipedia. 2003.
[7] Robert Larsson.
[8] Posten AB.

3.5 Ordutvigdning

En metod som används i stor omfattning är ordutvidgning. Metoden innebär att om en fras där ett antal intilliggande ord alla börjar på stor bokstav och ett av de ingående orden skall taggas, t.ex. för att det ligger i en databas, så taggas hela frasen. I exemplet med ”Drottning Silvias barnsjukhus” hittas först ”Silvia” i databasen varpå ordutvidgning sker och frasen ”Drottning Silvia” taggas med NAME-tagg. I det här fallet kommer vid ett senare tillfälle ännu en ordutvidgning och ett taggtypsbyte att ske så att hela frasen taggas som LOCATION.

3.6 Taggtypsbyte

I exemplet med ”Drottning Silvias barnsjukhus” taggas frasen genom en annan ofta använd metod, taggtypsbyte. I det här fallet hittas en ändelsen ”sjukhus” vilket innebär att frasen skall taggas med LOCATION. Sedan finns en regel som kontrollerar om intilliggande fras är i form av en NAME-tagg och i så fall ändras denna tagg till LOCATION. Denna form av taggtypsbyte då en ordändelse ligger efter ett personnamn förekommer också ofta för gatunamn och torg, t.ex. ”Per Albin Hanssons väg” och ”Gustav Adolfs torg”.

En annan form av taggtypsbyte sker utan ordutvidgning så att taggen endast byter namn. Detta sker då den taggade frasen förkommer i ett visst sammanhang, i en viss kontext, i meningen.

3.7 Kontextregler

De regler som hitills beskrivits har varit syntaktiska, reglerna har använt sig av de ingående ordens uppbyggnad utan att ta hänsyn till i vilket sammanhang frasen förekommer i meningen. Den typ av regler som istället utnyttjar detta kallas för kontextregler.

I frasen ”... utfarten på vägen Norra Åsum-Gärds Köpinge.” kommer ”Norra Åsum-Gärds Köpinge” först att taggas som CITY genom en kombination av ändelseregler och ordutvidgning. I sitt sammanhang i meningen kan man dock se att det rör sig om en väg och inte en stad. Nu kommer en kontextregel som utför ett taggtypsbyte på CITY-taggar som följer på nyckelordet ”vägen” att genomföras. Några exempel på kontextregler som används:

- Ord som följer på ”staden” eller ”förorten” taggas som CITY: ”i Parisförorten Créteil”, ”norr om turiststaden Flores”. Här taggas ”Créteil” och ”Flores” med CITY-taggar.
- Där någon förs till Xxx eller transporteras i någonting till Xxx och liknande konstruktioner används i en regel. I den här kontexten syftar Xxx på en plats och märks upp med LOCATION-taggar: ”transportera honom till Universitetssjukhuse MAS”, "fördes i ambulans till USÖ". I de här fallen tagna från vår testkorpus märks ”Universitetssjukhuse MAS” och ”USÖ” med LOCATION-taggar. På grund av felstavning kunde inte tidigare syntaxregler märka upp det första exemplet korrekt och sjukhuset ”USÖ” kunde inte heller finnas utan hjälp av denna kontextregel.
- Fraser med ordsekvenser där orden börjar på stor bokstav som följer på ord som ”säger”, ”berättar”, ”menade” o.s.v. taggas med hjälptaggen NAME: ”... sade Signe Lenstad ...”. Här taggas ”Signe Lenstad” med NAME-tagg.

3.8 Taggsammanslagning

I vissa sammanhang förekommer två taggar av samma typ intill varandra som egentligen tillhör samma entitet. T.ex. slås de två LOCATION-taggarna

```
<ENAMEX
TYPE="LOCATION">Universitets
sjukhuset</ENAMEX> <ENAMEX
TYPE="LOCATION">Mas</ENAMEX>
```

ihop till en LOCATION-tagg

```
<ENAMEX
TYPE="LOCATION">Universitets
sjukhuset Mas</ENAMEX>
```

3.9 Genererade databaser

Då alla nämnda regler implementerats på texten och hjälptaggarna tagits bort genereras en databas för LOCATION och CITY. Detta går till så att den taggade texten gås igenom och samtliga fraser som taggats med LOCATION- eller CITY-taggar läggs in i en databas. Därefter sker en slutlig taggning av texten med fraserna i den genererade databasen. Alla otaggade förekomster av dessa fraser taggas.

4 Implementering

Java var det naturliga språket att använda vid utvecklingen då CarSim-projektet[9] är javabaserat. Java har även ett regexpaket inbyggt vilket användes flitigt i programmet.

Programmet är uppbyggt så att texten märks upp sekventiellt med de olika taggarna. Sedan tas hjälptaggarna bort och en sista taggning sker med hjälp av den genererade databasen. Det visade sig att ordningsföjden i vilken de olika fraserna taggades var viktig.

4.1 Taggordning

I programmet sker taggning och övriga operationer i följande ordning: REMOVE → NAME → AREA → LOCATION → SQUARE → ROAD → HIGHWAY → STREET → CITY → LOCATION igen → STREET igen → Borttagning av REMOVE-, NAME- och AREA-taggar → generering av databas. Här ges några exempel för att motivera denna ordningsföljd:

- REMOVE → NAME

  REMOVE-taggar implementeras före NAME-taggar för att undvika att t.ex. en fras som "Peter Anderssons Volvo" taggas som NAME. I nuläget taggas frasen korrekt, "Peter Anderssons" som NAME och "Volvo" med hjälptaggen REMOVE (som tas bort i slutresultatet). Skulle ordningsföljden vara den omvända skulle hela frasen taggas som NAME p.g.a. utvigdningsregeln för NAME-taggarna.

- REMOVE → STREET, SQUARE och LOCATION

  REMOVE-taggar implementeras före STREET-, SQUARE- och LOCATION taggar för att kunna använda regler för utvigdning och taggtypsbyte då nyckelord följer på NAME-taggar i fraser som t.ex. ”von Rosens väg” och ”Drottning Silvias barnsjukhus”.

- NAME → CITY

  Huvudanledningen till att hjälptaggen NAME infördes var att ändelser på otsnamn ofta var desamma som ändelser för personnamn. Exempel som ”Olle Stenholm” – ”Laholm” är vanliga. Skulle ordningen vara den omvända skulle inte ändelseregler för ortsnamn kunna användas i lika stor omfattning och resultatet skulle bli sämre. Även om man skulle välja att ha kvar NAME-taggarna i slutresultatet är denna ordning den bästa. Ändelseregler är inte lika kritiska för att få bra resultat för personnamnstaggning som för ortsnamnstaggning.

- LOCATION → CITY

  Det inträffar mer frekvent att ett ortsnamn är en del av en namngiven plats, än tvärtom. Skulle taggordningen vara den omvända skulle t.ex. frasen ”Malmö Allmänna sjukhus” bestå av två taggar, ”Malmö” skulle taggas som CITY och ”Allmänna sjukhus” som LOCATION. Med användning av ordutvigdning taggas nu hela frasen korrekt som LOCATION.

- LOCATION → CITY → LOCATION igen

  LOCATION-taggning sker i två omgångar. I den andra omgången implementeras ett par kontextregler samt sammanslagning av intilliggande LOCATION-taggar. Det är intuitivt inte alldeles självklart varför denna ordning är den bästa men testresultaten på vårt testkorpus blev bäst med denna ordning. Som exempel implementeras här kontextregeln att ord med stor bokstav som följer på frasen ”i höjd med ” taggas som LOCATION. Då en annan taggordning testades blev resultatet sämre med denna regel och fler ord taggades felaktigt med CITY-taggar.

[9] Per Andersson, 2003.

- Borttagning av REMOVE-, NAME- och AREA-taggar → generering av databas

  Det visade sig att hjälptaggarna i en del fall användes felaktigt. Genom att göra den sista taggningen utifrån den genererade databasen efter borttagningen av hjälptaggarna kunde dessa feltaggningar i många fall repareras.

4.2 Svårigheter

Det finns ett antal anledningar till att ett platsuppmärkningsprogram aldrig kan bli 100%-igt. För det första kan aldrig alla fraser hittas med hjälp av syntaxregler då stavningen ej är regelbunden, det finns undantag från nästan alla ”regler”.

Att använda stora databaser är en hjälp men det är omöjligt att få dem tillräckligt stora och heltäckande. I vissa fall är delar av dem oanvändbara då det finns ett antal ord som har flera betydelser. Under projektet kom vi t.ex. i kontakt med en databas med ortsnamn där orter som ”Åsa”, ”Dorotea”, ”Maria” och ”Bara” förekom.

Det finns ett antal kontextregler som ger mer eller mindre bra resultat. I många regler är det enda man kan få ut att det rör sig om någon form av entitet. I den mycket vanliga konstruktionen ”mellan Xxx och Yyy” kan de båda objekten vara t.ex. bilar, städer, platser, länder eller landskap. Även för regeln ”i höjd med Xxx” gäller detta, men i det här fallet visade sig att en märkning med LOCATION förbättrade mätresultatet under förutsättningen att regeln applicerades i slutet av programexekveringen. Det finns en mängd hypotetiska kontextregler där man ibland får förbättrade mätvärden och ibland inte. Det är ofta omöjligt att förutsäga resultatet innan man testat på sin testkorpus. Vad som också kan vara vanskligt med dessa regler är att de kan försämra resultatet efter en modifiering av databaserna eller införande av nya regler trots att regeln förbättrade resultatet tidigare.

5 Utvärdering

5.1 Testvärden

Vokabuläret som används i utvärderingen är hämtad från texten “Named Entity Recognition without Gazetteers”[10]. Vi utgår från en otaggad text.

- **Answer file** = Texten, maskinellt taggad av vårt program.
- **Key file** = Texten, manuellt taggad. Denna text utgör definitionen på korrekt taggning.
- **Recall** = Antalet korrekta taggar i Answer file dividerat med det totala antalet taggar i Key file.
- **Precision** = Antalet korrekta taggar i Answer file dividerat med det totala antalet taggar i Answer file.

Utvärderingen görs genom att räkna ut precision- / recallvärden. Vi jämför vår program-taggade korpus (Answer file) med en korpus som vi taggat för hand (Key file).

Den otaggade texten vi utgår ifrån består av tio texter på svenska från tidningsartiklar om trafikolyckor och består av drygt 2000 ord. Urvalet av texterna skedde på måfå från en större korpus. Ingen av utprovningen av programmets regler har gjorts på denna text.

För utvärderingen använde vi ett perlscript som anger hur många ordgrupper som blev uppmärkta, hur många ordgrupper som skulle uppmärkas, hur många ordgrupper som blev korrekt uppmärkta samt recall- och precisionvärden. Nedan följer resultatet av en testkörning av programmet på testkorpusen.

```
Antal ordgrupper som blev
uppmärkta: 60
Antal ordgrupper som skulle
uppmärkas: 66
Antal ordgrupper som blev
korrekt uppmärkta: 58
Precision: 0.966666666666667
Recall: 0.878787878787879
```

Värdena i testen får anses vara bra med tanke på projektets ringa omfång. T.ex. taggade vårt program frasen:

[10] Andrei Mikheev, Marc Moens and Claire Grover. 1999.

```
”<ENAMEX
TYPE="CITY">Vanneberga</ENAM
EX> ängar”
```

vilket skulle taggas som

```
”<ENAMEX
TYPE="LOCATION">Vanneberga
ängar</ENAMEX>”
```

Anledningen till feltaggningen var att ändelsen ”ängar” inte fanns med i listan över LOCATION-ändelser (där t.ex. ”skola”, ”sjukhus” etc. finns).

I texten “Named Entity Recognition without Gazetteers”[11] som handlar om taggning av texter på liknande sätt som här presenteras en del testresultat. Det är intressant att se vilka värden ett program i närheten av ”state of the art” på områden får. De testade sitt program på ett testkorpus med en databas på 4900 platsnamn och 30 000 personnamn. Dock skiljer sig det testet från vårt på ett antal punkter:

- Texterna var på engelska.
- Texterna var mycket mer allmänna. Texterna som vårt program är optimerade för, bilolycksartiklar, är mycket smalare. Man inser att vår uppgift därför var lättare.
- Texterna var inte anpassade efter namnen i ett visst land. Det blir uppenbarligen mycket lättare om man, som i vårt fall, till övervägande del bara behöver ta hänsyn till svenska namn.
- Reglerna för taggning var inte identiska. Det som verkar mest likt var deras taggning av platsnamn. De presenterar separata värden för de olika taggtyperna.

I deras test fick de precision- / recallvärden på 94 / 95 på platsnamn. De gjorde även ett test på personnamn och fick då något högre värden. De gjorde sedan om testet utan databaser och fick 59 / 46 på platsnamn och 95 / 90 på personnamn. Då de istället använda betydligt mindre, men noga uttestade, databaser fick de värden omkring 90 / 90 på platsnamn. Vad gäller omfång på databaserna var de i detta fall mest lika våra.

[11] Andrei Mikheev, Marc Moens and Claire Grover. 1999.

Det skulle vara intressant att göra ett liknande test av vårt program med begränsningar i databasernas omfång. Tyvärr är våra databaser hårt integrerade i programmet så detta lät sig inte göras på ett enkelt sätt. Vi gjorde dock ett par andra tester för att utvärdera vilken effekt åtgärderna hade på resultatet.

I det första testet tog vi bort den genererade databasen för CITY- och LOCATION-taggarna och fick följande testresultat:

```
Antal ordgrupper som blev
uppmärkta: 58
Antal ordgrupper som skulle
uppmärkas: 66
Antal ordgrupper som blev
korrektuppmärkta: 56
Precision: 0.96551724137931
Recall: 0.848484848484849
```

Här är den enda skillnaden att två förekomster av staden ”Norje” i texten inte kunde hittas. Då en förekomst hittades av programmet klarade det i originalutförande av att tagga de övriga två förekomsterna med hjälp av den genererade databasen, vilket alltså inte gjordes här.

I nästa test inaktiverade vi samtliga hjälptaggar, REMOVE, NAME och AREA, från programmet och erhöll följande resultat:

```
Antal ordgrupper som blev
uppmärkta: 67
Antal ordgrupper som skulle
uppmärkas: 66
Antal ordgrupper som blev
korrektuppmärkta: 25
Precision: 0.373134328358209
Recall: 0.378787878787879
```

Här sjunker värdena rejält, programmet blir i det närmaste oanvändbart utan hjälptaggarna. Här visas ett par exempel på feltaggningar som uppkom:

```
“Signe
<ENAMEXTYPE="CITY">Lenstad</
ENAMEX>, 83 år,”
“säger
<ENAMEXTYPE="CITY">Jan</ENAM
EX> <ENAMEX TYPE="CITY">Olov
Källström</ENAMEX>,”
```

```
“husvagnssemester på
<ENAMEXTYPE="LOCATION">Öland
</ENAMEX>”
“vägsträcka i
<ENAMEXTYPE="CITY">Blekinge<
/ENAMEX>.”
```

5.2 Möjliga förbättringar

Här beskrivs några metoder som eventuellt kunde medföra bättre testresultat men som p.g.a. tidsbrist ej har implementerats eller testats.

I texten “Named Entity Recognition without Gazetteers”[12] finns ett förslag på hur man på ett bättre sätt hanterar ord som inleder en mening. För varje ord som inleder en mening, och alltså inleds med en stor bokstav, kontrolleras om ordet även förekommer inne i texten men med en liten bokstav i början av ordet. Finns ordet där är det sannolikt att ordet inte är en plats som skall märkas upp. I annat fall appliceras de vanliga reglerna på ordet. Man kan utveckla denna idé och införa en ordlista med så många av implementeringsspråkets ord som möjligt som inte är av de sökta typerna och göra motsvarande kontroll.

En idé vi funderat på är införandet av globala genererade databaser. Dessa skulle fungera på samma sätt som de genererade databaser vi använder oss av för tillfället fast de skulle sparas undan mellan varje programexekvering. På detta sätt skulle programmets databaser hela tiden växa. Utan att ha testat idén ansåg vi att det kunde finnas problem med denna metod. Vid varje programkörning genereras nya felaktiga eller dubbeltydiga ord och med tiden skulle databasen innehållas stora mängder felaktiga poster. För att metoden skall vara bra krävs antagligen att databaserna justeras manuellt mellan programkörningarna.

Vad som naturligtvis skulle förbättra värdena för programmet skulle vara att modifiera databaserna med fler / ”bättre” ord och prova ut fler kontext- / syntaxregler. Det finns hela tiden fler små förbättringar man kan göra på programmet, bara man har tillräckligt med tid. En förutsättning för detta är dock att man har tillräckligt stora träningskorpus.

Acknowledgement

Tack till Richard Johansson för stöd, idéer och perlscript.

Referenser

Andrei Mikheev, Marc Moens and Claire Grover. 1999. *Named Entity Recognition without Gazetteers*, HCRS Language Technology Group, University of Edinburgh. http://www.ltg.ed.ac.uk/~mikheev/papers_my/eacl99.ps

Per Andersson. 2003. *A Prototype to Extract and Visualize Information from Car Accident Reports in Swedish*, http://www.efd.lth.se/~d98pan/rapport.pdf

Posten AB. *Databas med 1575 svenska ortsnamn*, http://www.lysator.liu.se/runeberg/words/ord.ortsnamn.posten

Robert Larsson. *Databas med Sveriges 1000 vanligaste för- och efternamn*, http://rl.se/tusen.html

Susning.nu 2003. *Bilmärken*, http://www.susning.nu/Bilm%e4rke

Teknikens Värld. 2002. *Teknikens Värld begagnatprislista*, http://www.teknikensvarld.com/webbm2/teknik/teknik.nsf/d4f8e3df8882ba79c125687b004ebf65/b0a1372650c21952c1256c1e002c2817

Wikipedia. 2003. *Alfabetisk lista över världens länder*, http://sv.wikipedia.org/wiki/Alfabetisk_lista_%F6ver_v%E4rldens_l%E4nder

[12] Andrei Mikheev, Marc Moens and Claire Grover. 1999.

Beslutsträdsinduktion för probabilistisk taggning

Johan Enell
johan@enell.nu

Fredrik Larsson
dat98fla@ludat.lth.se

Sammanfattning

I detta projekt har vi granskat och implementerat en taggnings metod som är en utveckling av probabilistisk taggning. Skillnaden från den probabilistiska taggnigen är att transitions-sannolikheten uppskattas med hjälp av ett beslutsträd. Fördelen med detta är att man kan få en hög precision från en liten mängd exempel data. Ett problem som vi upptäckte var att skapandet av trädet tog väldigt lång tid. Av den anledningen utvecklade vi även ett annat sätt att skapa trädet på som avsevärt förbättrade den tidsåtgången. Enligt artikeln som vi fick algoritmen från var precisionen högre än det konventionella metoderna. Vi har inte kunnat testa om detta stämmer och vi har inte heller kunnat testa om våran metod fungerar eftersom det inte funnits tid att implementera de delarna av programmet.

1 Inledning

Att bestämma satsdelar på ord är inte helt trivialt eftersom dess betydelse kan vara ambivalent beroende på vilket sammanhang det befinner sig i. För att klara av detta brukar man titta på det sammanhang som ordet befinner sig i för att lista ut ordets rätta satsdel.

Det finns många olika metoder för att lösa detta problem på olika sätt. Vissa använder något form av regelsystem andra använder probabilistiska metoder. Även neurala-nätverk används för att lösa problemet. Om man tittar närmare på de probabilistiska metoderna så ser man att alla använder någon form av markovmodell. Eftersom det blir en stor mängd parametrar när man ska bestämma satsdelar (speciellt när man använder trigrams) blir det svårt för dessa metoder att det svårt för dessa metoder att uppskatta små sannolikheter.

I den algoritm som vi har jobbar med används ett beslutsträd för att uppskatta transitonssannolikheten.

2 Probabilistisk taggning

TreeTagger algoritmen grundar sig på den välkända och vanligt förekommande ngram taggeralgoritmen. Båda två modellerar sannolikheten av en taggad sekvens av ord rekursivt med formeln:

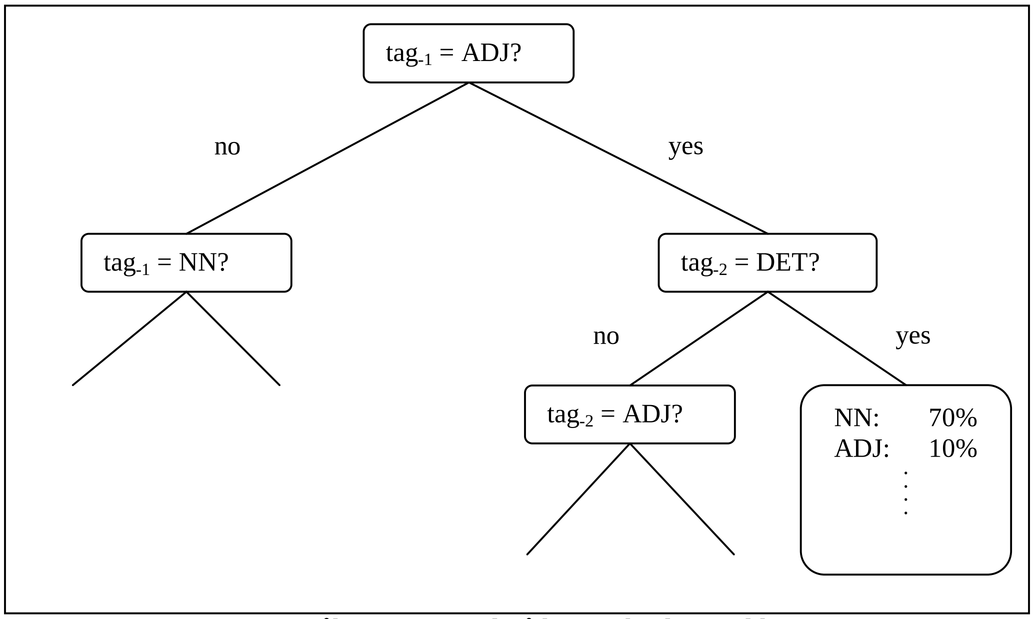
$$\begin{aligned} &p(w_1 w_2 \ldots w_n, t_1 t_2 \ldots t_n) := \\ &p(t_n | t_{n-2}, t_{n-1}) p(w_n | t_n) \\ &p(w_1 w_2 \ldots w_{n-1}, t_1 t_2 \ldots t_{n-1}) \end{aligned} \qquad (1)$$

Denna metod skiljer sig på hur transitonssannolikheten, $p(t_n | t_{n-2}, t_{n-1})$, uppskattas. För det mesta använder ngram-taggers en formel som baserar sig på ”maximum likelihood estimation (MLE)” principen:

$$p(t_n | t_{n-2}, t_{n-1}) = \frac{F(t_{n-2}, t_{n-1}, t_n)}{F(t_{n-2}, t_{n-1})} \qquad (2)$$

$F(t_{n-2}, t_{n-1}, t_n)$ är antalet förekomster av trigrammet t_{n-2}, t_{n-1}, t_n i korpuset och $F(t_{n-2}, t_{n-1})$ är antalet förekomster av bigrammet t_{n-2}, t_{n-1}.

Ett stort problem med denna uppskattning är att många frekvenser är så små så den sannolikhet man får fram inte är tillräckligt trovärdig. Det blir speciellt svårt i de fall då frekvensen är noll och man kan inte avgöra om motsvarande trigram är syntaktiskt korrekt eller om det bara är väldigt ovanligt. En annan viktig punkt är att en robust tagger ska kunna klara indata som inte är helt grammatiskt korrekt. Annars kan det inkorrekta



Figur 1: Ett påhittat exempel på hur ett beslutsträd kan se ut.

leda till att hela uttryck blir tilldelade noll i sannolikhet, oberoende av sekvensen av taggar. Detta skall undvikas.

Därför modifieras formeln ovan så att nollsannolikheterna ersätts med ett litet tal och seden normaliserar man om sannolikheterna så att summan blir 1. Valet av det tal som skall användas är väsentligt för kvalitén på resultatet.

3 TreeTagger

TreeTagger använder sig, till skillnad från ngram tagger, av ett binärt beslutsträd för att uppskatta transitions-sannolikheter. Figur 1 visar ett exempel på hur ett sådant träd ser ut. Sannolikheten för ett givet trigram fås genom att följa dess väg genom trädet tills man kommer till ett löv. Om vi t.ex. tittar på sannolikheten för att ett substantiv kommer efter en determinant som i sin tur kommer efter ett adjektiv p(NN| DET, ADJ) måste vi först svara på frågan i roten; Är taggen på position -1 ett adjektiv? I detta fall är svaret ja, så då går vi vidare till nästa nod via ja-vägen. Där svarar vi på frågan om taggen på plats -2 är en determinant. Även detta är sant så då kommer vi ner i ett löv. Där kan vi se att sannolikheten för detta trigram är 70%.

3.1 Skapa ett beslutsträd

Beslutsträdet som byggs från en träningsmängd av trigram använder en modifierad version av ID3-algoritmen. I varje rekursionsteg skapas ett test som delar mängden av trigram två delmängder. Delmängderna väljs så att skillnaden i sannolikhetsdistributionen för den tredje taggen blir så stor som möjligt. Testet undersöker en av de två framförliggande taggarna och kontrollerar om det är identiskt med en tag t i en mängd av taggar. Testet ser ut på följande sätt:

$$\begin{aligned} &tag_{-i} = t \\ &i \in \{1,2\} \\ &t \in T \end{aligned} \tag{3}$$

där T är en mängd av taggar.

Vid varje rekursionsteg jämförs alla möjliga test och det test som ger mest information tilldelas till nuvarande nod. Sedan utvidgas träder rekursivt på de två delmängderna som skapats utav testet. De träd som skapas läggs till som ja- och nej- träd på den nuvarande noden.

Det värde som används för att jämföra testerna, q, är den mängd information som fås av den tredje taggen när testet är utfört. Att maximera informationen är ekvivalent med att minimera genomsnittet av den information, I_q, som krävs för att den tredje taggen går att identifiera efter att resultatet av test q är känt.

$$I_q = -p(C_+|C)\sum_{t\in T} p(t|C_+)\log_2 p(t|C_+) - p(C_-|C)\sum_{t\in T} p(t|C_-)\log_2 p(t|C_-) \quad (4)$$

Där C är de trigram som finns i exempelmängde för den aktuella noden. C_+ är de trigram där test q lyckas och C_+ är de trigram då testet misslyckas. $p(C_+|C)$ är sannolikheten att testet q lyckas, motsvarande för $p(C_-|C)$. Slutligen är $p(t|C_+)$ sannolikheten för den tredje taggen om test q lyckas och $p(t|C_-)$ om det misslyckas. Dessa sannolikheter uppskattas med MLE:

$$p(C_+|C) = \frac{f(C_+)}{f(C)} \quad (5)$$

$$p(C_-|C) = \frac{f(C_-)}{f(C)} \quad (6)$$

$$p(t|C_+) = \frac{f(t,C_+)}{f(C_+)} \quad (7)$$

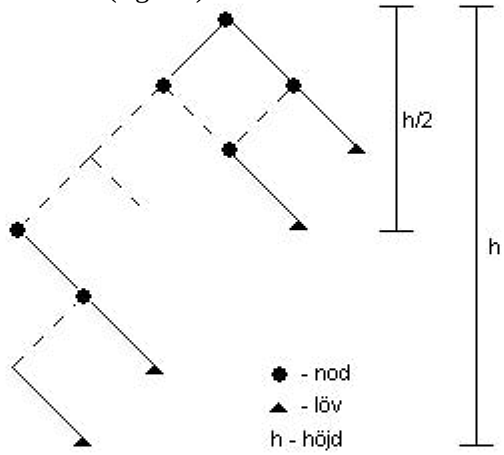
$$p(t|C_-) = \frac{f(t,C_-)}{f(C_-)} \quad (8)$$

Där $f(C)$ är antalet trigram i träningsmängden för den aktuella noden. $f(C_+)$ är antalet trigram där testet lyckas, motsvarande för $f(C_-)$. $f(t,C_+)$ är antalet trigram som klarade testet och där tredje taggen är t. Den rekursiva expansionen av trädet stannar om nästa test genererar en delmängd av trigram som är mindre än ett förutbestämt tröskelvärde, n ($f(C_+) < n$ eller $f(C_-) < n$). Sedan uppskattas sannolikheten $p(t|C)$ för den tredje taggen från de trigram som kommit till denna nod och värdet sparas i lövet.

$$p(t|C) = \frac{f(t,C)}{f(C)} \quad (9)$$

4 FastTree

När man tänker efter hur trädets form kommer att kunna tänkas vara inser man att det i princip alltid kommer att se likadant ut, och att man inte kan göra så mycket åt det. Med vänsterbarn som nej (ja), högerbarn som ja (nej) och att två ja ger löv resulterar alltid i ett träd som kommer luta starkt åt vänster (höger), med ett större antal noder. Vid ett fullt utvecklat träd med alla trigram-kombinationer kommer det nedersta lövet av högerdelen av trädet alltid ligga på halva djupet av det nedersta lövet i vänsterdelen (figur 2).



Figur 2: En enkel skiss av hur trädet kommer att se ut.

Tvärt emot vad namnet antyder är inte FastTree snabbare än något annat på att användas. Namnet försöker istället säga att den skapar träd betydligt snabbare än ID3.

Tanken med FastTree är enkel och metoden likaså. Eftersom trädet alltid, i princip, kommer ha samma form saknar balansering vikt vid skapandet av trädet och minskar därmed beräkningarna.

FastTree bygger istället helt på förekomster av unigram och bigram. Det vanligaste unigrammet väljs som rot i trädet, det näst vanligaste som vänsterbarn och så vidare. Högerbarnet är den vanligaste taggen med föräldern som föregående tagg, d.v.s. det vanligaste bigrammet med föräldern som första tagg i bigrammet. Dess vänsterbarn är den näst vanligaste taggen med den förra föräldern som föregående tagg. Ett andra högersteg resulterar i ett löv då vi har tre taggar som utgör ett trigram; vilket är vad vi söker.

På detta sätt bygger FastTree ett snabbt, korrekt, men förmodligen inte ett optimalt träd.

5 ID3 vs. FastTree

Problemet med ID3-baserade träd är uppenbar. För stora mängder träningsdata kan skapandet av trädet ta olämpligt lång tid. Algoritmen vi har implementerat har en logaritmisk tidskonsumtion med avseende på storleken av korpusen. Det hjälper dock föga då varje steg tar allt för lång tid och att ökande av antal olika tagg-typer ger en exponentiell tidsökning.

Nackdelen med FastTree är att det förmodligen skapar ett sämre träd. När algoritmen väljer nod vid yttersta nivån, dvs. Utan några ja-svar, tar trädet inte hänsyn till hur underträden kommer att se ut. Det är då möjligt att trädet som genereras av FastTree-algoritmen är genomsnittligt sämre än ID3-trädet vid sökning. Men eftersom det fortfarande är korrekt och att det är den totala tidsåtgången som spelar roll kan FastTree vara ett alternativ då man inte har ett färdigt träd att tillgå.

Using Speech Recognition for controlling a Pan-Tilt-Zoom Network Camera

Enrique Garcia
Department of Computer Science
University of Lund
Lund, Sweden
enriqueg@axis.com

Sven Grönquist
Department of Computer Science
University of Lund
Lund, Sweden

Abstract

In this paper we describe a prototype component for using speech to control a remote controllable a web camera.

We investigate some situations where speech control might be useful, and describe the results of testing the component.

Our conclusion is that speech control can be a useful complement to the traditional traditional point and click interface.

1 Introduction and Background

In this project we developed a Speech Recognition component for controlling a Pan-Tilt-Zoom Network Camera. This report describes the speech recognition background, the details of our implementation and its evaluation.

1.1 Speech Recognition

Speech Recognition is a technology that allows a computer to identify the words that a person speaks. Traditionally, the input devices to the system are microphone and/or telephone.

The history of speech recognition goes back to the 19th century when Alexander Graham Bell tried to build a machine that could recognize the human voice [1]. In 1950 Bell Laboratories could build a machine that recognized the ten digits. Later in 1954 MIT developed the first hardware for identifying vowels. In the 1970s DARPA established a program for understanding continuous speech.

In the last years, dramatic improvement in speech recognition technology has taken place. This is due to new research and better algorithms, as well as computers with more processing power.

Today several methods and technologies are involved in speech recognition. The Hidden Markov Model is one example of a widely used statistical method that is based on the idea that the speech signal can be characterized as a parametric random process. Template methods use average procedures to derive words and a local spectral distance measure to compare patterns.

Despite the dramatic improvement in technology, further research is still needed to cope with the limitations of speech recognition. Some of the most important limitations are: Speech recognition systems are easy to confuse when using a large vocabulary, they find it difficult to differentiate between short words, and a high accuracy (up to 95%) is only achieved in quiet environments. Today the best way to handle some of these limitations is to train the system and learn the user to operate it.

There are many advantages using speech recognition, besides of the obvious reason that it is a more natural way to interface numerous computer applications than using a mouse and a keyboard, speech recognition allows faster input, and offers the users great freedom of mobility; hands and eyes are free.

Users who could benefit from speech recognition are people who for some reason are unable to (or find it difficult to) use a normal keyboard or mouse (*e.g.* people with hand or eye problems). It might also be useful to professionals producing written reports, but who traditionally don't type themselves, like doctors, psychologists and lawyers. On the other hand using speech recognition for such reports requires an extremely high accuracy.

Additionally speech recognition technology is applied in telephone networks to automate operators' services and it is found in Personal Digital Assistants (PDAs), mobile phones and other client devices where keyboard input is difficult.

During the last years speech recognition has begun to be used for Internet applications. VoiceXML [2] and Speech Application Language Tags (SALT) [3] are two standards for navigating web pages and access remote database by speech only. Companies such as Microsoft, IBM, Philips and ScanSoft have presented working solutions based on these standards.

Especially Microsoft has introduced support for speech recognition in the .NET platform and a plug-in for Internet Explorer [4] that allows users to browse SALT enhanced web pages. Microsoft has also for years offered a freely downloadable Speech SDK that includes a very good speech recognition engine, components, samples and tutorials.

1.2 Pan-Tilt-Zoom (PTZ) Network Camera

A network camera can be described as a camera and a computer combined. It is a camera connected directly to the network. Inside the camera it is a CPU, Linux OS, a web server etc. A network camera has its own IP address and built-in functions to handle network communication. [5]

Everything needed for viewing images over the network is built into the unit. An embedded web server manages web pages for displaying the video and handles different HTTP requests for controlling the camera and displaying the video via Internet/Intranet, see figure 1.

Network cameras are used in professional security systems for surveillance of sensitive areas, such as buildings, casinos, banks and shops. Video of those areas can be monitored from control rooms, at police stations and by security managers from a variety of locations.

A Pan-Tilt-Zoom Network Camera is a network camera that allows the users to rotate it horizontally and vertically (Pan, Tilt) and also to adjust its zoom level (Zoom).

An example of a PTZ Network Camera is the AXIS 2130 (figure 2), developed by Axis Communications AB, located in Lund, Sweden. More about this PTZ Network Camera can be found at [6].

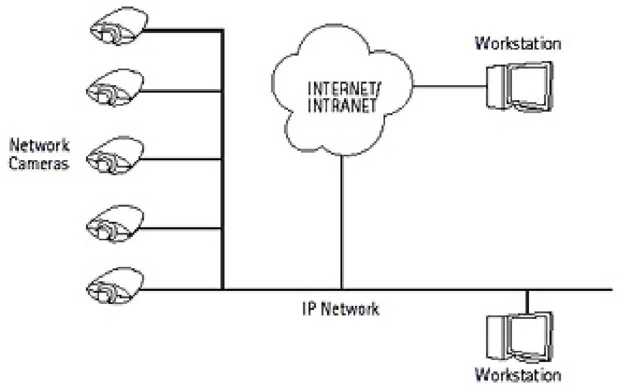


Figure 1: Network Cameras are able to transmit video and being controlled through Internet.

Figure 2: AXIS 2130 PTZ Network Camera. Here showing three different positions

The camera is controlled by client software using `HTTP GET` requests that are sent to the camera's web server. These requests are defined in an application programming interface (API) which is independent of the kind of network camera.

For example for turning the camera with IP address `10.13.5.35`, five degrees to the left, the client software issues following request to the camera:

```
HTTP GET http://10.13.5.35/axis-cgi/
ptz.cgi?move=left
```

More information about the API and a link to the cammand details can be found in Appendix A – Axis PTZ API.

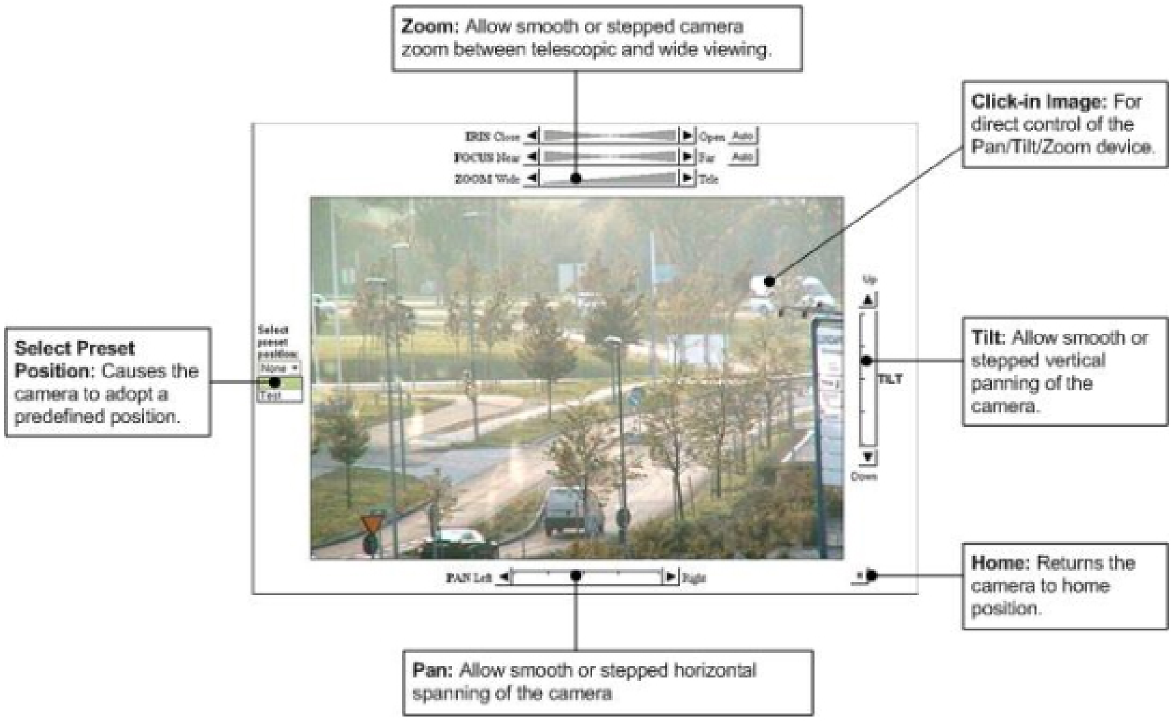

Zoom: Allow smooth or stepped camera zoom between telescopic and wide viewing.
Click-in Image: For direct control of the Pan/Tilt/Zoom device.
IRIS Close
Open
Auto
FOCUS Near
Far
Auto
ZOOM Wide
Tele
Up
TILT
Down
Tilt: Allow smooth or stepped vertical panning of the camera.
Select Preset Position: Causes the camera to adopt a predefined position.
Select preset position
None
Test
PAN Left
Right
Home: Returns the camera to home position.
Pan: Allow smooth or stepped horizontal spanning of the camera

Figure 3: AXIS 2130 Client GUI

The AXIS 2130 client's graphic user interface (GUI) for controlling the PTZ camera is based on mouse control and shown in figure 3.

The users click the directions buttons with the mouse and these clicks are translated to the corresponding requests using the PTZ API.

2 Problem Statement

The project has as objective to implement and evaluate a speech recognition component for controlling the AXIS 2130 PTZ Network Camera.

Using speech recognition, the component will translate voice commands to PTZ requests managing the camera's positioning and zoom level.

We would like to test and compare this approach with the standard PTZ GUI, driven by mouse clicking in a gradient bar and directions buttons and see if the speech driven user interface could offer any advantages like natural user interaction, lower learning factor, commodity and quicker reactions of the users controlling the camera.

We believe that the results could motivate further development of new products that combine speech recognition and network camera technologies for enhancing the user experience.

3 Material and Methods

3.1 Defining Use Cases

We have identified the following scenarios. Here we don't look at a particular camera's capabilities, but rather what one might want to achieve.

General commands

The most obvious set of basic commands that we need to implement are the simple movement commands like '`up`', '`down`', '`left`', '`right`', '`undo`' etc. More complicated commands could be for moving at a certain speed (in pan and tilt directions), and even to 'lock on' a specific (moving) object. The more specific scenarios can then use these as primitives.

Another set of commands could be for handling preset positions such as '`record position N`' and '`go to position N`'. Ideally the they should be named (gate, coffee machine, etc), but we could use numbered presets.

For debugging purposes we might want additional commands such as '`reload grammar`' or '`start` `logging`'.

Surveillance

One operator uses one or more cameras, and should be able to control any one camera at a given time. Here we have the following problems:
– More than one camera: We must give each a unique name, or, probably easier, a number. The operator shouldn't need to address a camera with its IP number. Using names is preferable, but names must not conflict with the commands (i.e. cameras should not be named '*left*' and '*right*').
– Only one camera can be operated at a given time. We need commands like '`activate camera N`' and perhaps '`deactivate camera N`' (but activating one camera should deactivate the others). These cammands could be mouse/keyboard-operated.
– We should consider the difference between a system where a camera only has a number of fixed presets (show gate, show machine A, show machine B etc), and a more *ad hoc* system where the operator can do anything.
– Sequences of operations, letting the user (operator?, manager?, *cf* web camera below) define a number of 'targets' (gate, machine A etc) in terms of pan-tilt-zoom, and letting the camera 'visit' each target. In this way one camera can work in a semi-autonomous mode, while the operator works with other things. This might get quite complex, the operator should be able to stop the camera, look at something particular, and then letting it resume.
– We must have a useful and consistent interface for working when speech control is not possible (e.g. when the operator is using the telephone). Sometimes it might be better *not* to use speech control.

Also: For a system like this it is acceptable to train the system for the operators voice (altough our solution doesn't do that).

Exhibitions

Additional issues for using the system at an exhibition:
– Working in a noisy environment (specifically lots of talk around),
– handling unknown voices (again, we don't use a training system, so this is not a problem here).

Web camera (or Web Attraction)

Controlling a web camera raises two new problems:
– Setting up more than one role. The first role is the *owner*, the person who sets the various targets (like '`coffee machine`', '`window view`' etc). The other is the *user* who should be able to say '`show coffee machine`' etc, but not much more.
– handling more than one user. When more than one user tries to control the camera at the same time ther might be synchronizing problems.
We don't investigate this any further.

3.2 Using the Microsoft Speech SDK

In this project we have chosen to use the Microsoft Speech Software Development Kit (SDK) because of the simplicity of the application programming interface (API) and the good support that can be found in this free package.

We avoided using the most recent Microsoft .NET SDK, because of the required upgrades in many components and other unavailable tools. (such as Microsoft Visual Studio .NET 2003).

The Speech SDK v 5.1 comes with COM, Active-X, VB and VC++ interfaces for speech recognition and speech synthesis for Windows. It also includes freely distributable text-to-speech engine (TTS) and speech recognition engine for U.S. English.

Using the Speech SDK is straightforward, a number of tutorials explain how to initialize the speech recognition engine and how to define the grammars for using it.

More information about the Microsoft Speech SDK is available at [6].

4 Prototype

4.1 A Speech Recognition Active-X Control

We developed an Active-X control which can recognize the general commands for controlling the camera. Building the prototype as an Active-X control makes it possible to embedded it on a web page below the Axis client that displays the video.

The GUI has been kept simple showing the available commands. When a command is identified that command's text color changes to red (see figure 4).

We wrote a simple grammar defining the following commands:

```
[Show me|Go to] (the) [left|down|right|up|home]
```

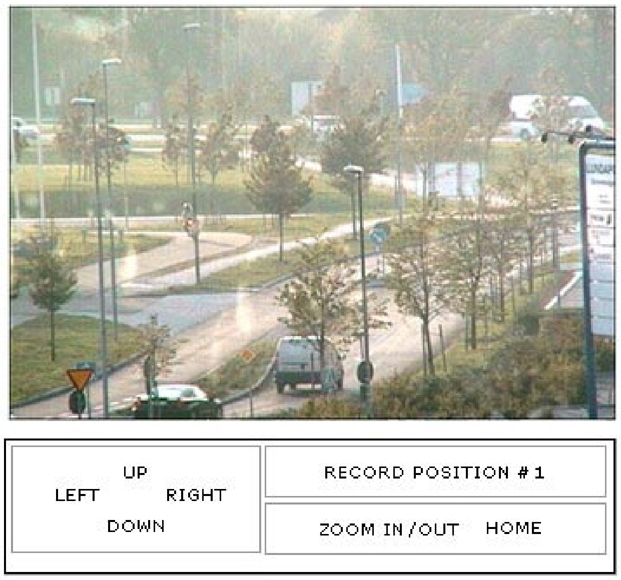


Figure 4: The prototype GUI

`[Zoom] [in|out]`
`[Record|Show me] [one|two|three]`

The last one is for recording and displaying the preset positions. In that way the prototype can handle the use cases for simple commands and the surveillance case for predefined presets.

5 Results and Discussion

We have tested the prototype with three subjects. We had prepared the following tests:

1. Rotate the camera so that a predefined object will be displayed in the center of the video (pan and tilt operations were necessary to center it)
2. Follow the trajectory of an object (a person walking).
3. Zoom in to a distant object so that it covers the entire display.
4. Record and view three different preset positions.

The tests raise a number of difficulties: using simple commands are not enough for exactly positioning the camera, it was also necessary in some cases to repeat the command many times and did show that others words like '`more`', '`little more`', '`stop`', '`back`' and '`again`' should also be handled like commands.

Any time that a subject used for example he word '`left`' the camera turned on the left exactly five degrees, sometime much more or much less than necessary. Positioning exactly the camera without specifying the degrees seems difficult as to imagine how many degrees are necessary. The GUI based in mouse click gives a scale that it is more accurate and clicking somewhere in the picture for centering the camera seems simpler than using the speech version.

A collected interaction in the test 1 follows:

```
- turn on left
- more
- more to the left
- more
- more
- more to left
- stop
- turn little to the right
- right
```

'`Stop`' is not supported neither by the camera or the prototype. '`Little`' was not supported so the user found it more difficult to complete the task.

The most successful test was about recording and viewing the presets. Users found it useful, and easy to use. They state that it should be possible to name each preset by speech and then call them instead of, as now, using numbers.

During the tests the accuracy was very good and the engine proved to be good enough for commands.

6 Conclusions

The exactly positioning of the camera seems difficult using speech, as well as following moving objects. Our simple commands don't allow accurate positioning for the pan-tilt-zoom values.

Our prototype seemed more useful when using presets positions.

6.1 Future directions

Furter development of the prototype should investigate other approaches like for example using scales on the side of the video. We think the accuracy will increase but it will never be as simple as using the mouse. Probably speech control is most suited to situations where the users are unable to use the mouse or the keyboard (or they are used for other purposes).

Future version should allow renaming the position and maybe interacting with the user in a dialogue about the position presets available.

Other camera client functions that can be easy mapped to commands like ‘`Take a shot`’, ‘`Show Full Screen`’ and more could be added.

Networks Cameras can also be accessed using a PDA, that field should also be investigated at least with preset positions in mind.

7 Acknowledgements

We would like to thank Pierre Nugues and Mathias Haage at the Department of Computer Science at Lund University for valuable insights and comments during our work.

References

1. History of Speech Recognition
   `http://nexus.carleton.ca/~kekoura/history.html`
2. VoiceXML 2.0 Last Call
   `http://www.w3.org/TR/voicexml20/`
3. SALT Forum
   `http://saltforum.org/`
4. MS Speech SDK
   `http://microsoft.com/speech/`
5. What is a Network Camera?
   `http://www.axis.com/products/video/video_clip/axiscam_448x336_mpeg1_high.mpg`
6. AXIS 2130 PTZ Camera
   `http://www.axis.com/products/cam_2130/index.htm`

Appendix A – Axis PTZ API

The complete Axis Camera API, HTTP - Interface Specification v 1.11 can be found at: `http://www.axis.com/techsup/cam_servers/dev/cam_http_api.htm`. Only the Pan/Tilt/Zoom cameras support the PTZ commands.

To control the Pan, Tilt, and Zoom behavior of a PTZ unit, the following PTZ control URL is used.
Method: GET/POST
Syntax: `http://<servername>/axis-cgi/com/ptz.cgi?<parameter>=<value>[&<parameter>=<value>...]`
with `parameters` and `values` from the Interface Specification.

Appendix B – PTZ Grammar

```
<GRAMMAR LANGID="409">
  <DEFINE>
    <ID NAME="VID_Left" VAL="10"/>
    <ID NAME="VID_Right" VAL="20"/>
    <ID NAME="VID_Up" VAL="30"/>
    <ID NAME="VID_Down" VAL="40"/>
    <ID NAME="VID_Home" VAL="50"/>

    <ID NAME="VID_One" VAL="60"/>
    <ID NAME="VID_Two" VAL="70"/>
    <ID NAME="VID_Three" VAL="80"/>

    <ID NAME="VID_Place" VAL="90"/>
    <ID NAME="VID_Navigation" VAL="100"/>

    <ID NAME="VID_Position" VAL="110"/>
    <ID NAME="VID_Record" VAL="120"/>
    <ID NAME="VID_Number" VAL="130"/>

    <ID NAME="VID_Zoom" VAL="140"/>
    <ID NAME="VID_Direction" VAL="150"/>
    <ID NAME="VID_In" VAL="160"/>
    <ID NAME="VID_Out" VAL="170"/>
  </DEFINE>

  <RULE ID="VID_Navigation" TOPLEVEL="ACTIVE">
    <O>Please</O>
    <P>
      <L>
        <P>Show me</P>
        <P>Go to</P>
      </L>
    </P>
    <O>the</O>
    <RULEREF REFID="VID_Place" />
  </RULE>

  <RULE ID="VID_Place" >
    <L PROPID="VID_Place">
      <P VAL="VID_Up">up</P>
      <P VAL="VID_Right">right</P>
      <P VAL="VID_Left">left</P>
      <P VAL="VID_Down">down</P>
      <P VAL="VID_Home">home</P>
    </L>
  </RULE>

  <RULE ID="VID_Record" TOPLEVEL="ACTIVE">
    <O>Please</O>
    <P>
      <L>
        <P>Record</P>
      </L>
    </p>
    <RULEREF REFID="VID_Number" />
  </RULE>

  <RULE ID="VID_Position" TOPLEVEL="ACTIVE">
    <O>Please</O>
    <P>
      <L>
        <P>Show me</P>
      </L>
    </p>
    <RULEREF REFID="VID_Number" />
  </RULE>

  <RULE ID="VID_Number" >
    <L PROPID="VID_Number">
      <P VAL="VID_One">one</P>
      <P VAL="VID_Two">two</P>
      <P VAL="VID_Three">three</P>
    </L>
  </RULE>

  <RULE ID="VID_Zoom" TOPLEVEL="ACTIVE">
    <O>Please</O>
    <P>
      <L>
        <P>Zoom</P>
      </L>
    </p>
    <RULEREF REFID="VID_Direction" />
  </RULE>

  <RULE ID="VID_Direction" >
    <L PROPID="VID_Direction">
      <P VAL="VID_In">in</P>
      <P VAL="VID_Out">out</P>
    </L>
  </RULE>
</GRAMMAR>
```

A spell checker with a user model for Swedish dyslexics

Marie Gustafsson
Department of Computer Science
Lund University
Sweden
marie@katastrof.nu

Abstract

Dyslexics pose a great challenge to spelling checking programs. They are among the ones who need the programs the most, they make diverse and complicated errors, and they may have trouble picking out the intended word from a long list of suggestions. The idea behind the program described in this article is to start with a spell checker based on a noisy channel model and allow for multiple transformations of deletion, insertion, substitution and reversal between the typo and the intended/suggested word. A user model consists of matrices of how often the user makes these transformations for the different letters of the alphabet. When the user chooses a suggestion from the spell checker, the typo/correction pair is used to further update these matrices.

1 Introduction

Dyslexia is a subject of much debate, both regarding definition and diagnosis. Because spelling is a large burden for dyslexics, most agree that a spell checker can be of great advantage. Alas, most spell checkers are designed for correcting typing errors, or typos, made by fairly able spellers. However, dyslexic spellers make more diverse errors, including compound errors consisting of a sequence of mistakes. Many dyslexics experience that spell checkers are not able to suggest words for their misspellings. Further, a small number of suggested corrections is important, since it may be difficult for the dyslexic to select from a long list (Spooner and Edwards, 1997).

A computer can have a user model for predicting how a user thinks and behaves. Ordinary spell checkers have minimal if any user model. It is thought that a user model can improve a spell checker by being more tuned to the kinds of mistakes that a certain user tends to make. For example, one user might tend to confuse b and d, while another might have problems with p and b. Knowing about these individual confusions can improve the suggestions given by the spell checker.

This article will look at the possibilities of applying a simple user model to a spelling correction program presented by Kernighan et al. (1990) , which is based on a noisy channel model. More specifically, the possibilities of applying this user model to Swedish dyslexics.

2 Dyslexia

The definition of dyslexia has been much discussed, as well as whether or or not dyslexia should be defined at all. Many definitions have focused on a discrepancy between the ability to read and write and the other intellectual abilities of a person. In 1994, the Orton Society decided on the following definition(Hoien and Lundberg, 1999):

> Dyslexia is one of several distinct learning disabilities. It is a specific, language-based disorder of constitutional origin characterized by difficulties in single word

decoding, usually reflecting insufficient phonological abilities.

The causes of dyslexia are disputed, but research has shown that phonological training in early schooling can be helpful. While many dyslexics eventually manage to read at a fairly high level, the troubles with spelling are more persistent.

3 User modeling

A user model defines the way a computer believes a person using it will behave. Dynamic user model generally refers to a set of stored numbers indicating how a particular person behaves on a number of scales. The field of user modeling has been around for about twenty years, starting with student modeling in the early eighties. The cognitive processes that underlie the user's actions and the user's behavioral patterns or preferences are some things that user models may wish to describe. Another is the difference between the user's skills and expert skills (Webb et al., 2001).

Among others, the following structures and processes are often included in a user modeling system (Kobsa, 2001):

- the representation of assumptions about user characteristics in models of individual users, such as assumptions about knowledge, misconceptions, goals and preferences;
- the representation of common characteristics of users, grouping them into subgroups, or stereotypes;
- the classification of users as belonging to one or more subgroups, along with the integration of typical characteristics of these subgroups into the current individual user model;
- the recording of users' behavior, especially their past interaction with the system;
- the formation of assumptions about the user based on the interaction history.

Observing the user's behavior can provide examples for training the user model, which can be used to make a model to predict future actions. There are however problems: the need for large data sets; the need for labeled data; concept drift; and computational complexity (Webb et al., 2001). Further, if user modeling profiles are to be created, a sufficient amount of time is required before they can be of any use. A solution to this might be the incorporation of stereotypes. Ideally, these should be used for initializing the user model, until there is more information about the individual user. Also, it should be regularly checked whether or not the right stereotype is activated (Virvou and Kabassi, 2002).

4 Approaches to spell checking

The traditional spell checker will go through a text and for each word check if it is in its dictionary. If it is, the spell checker moves on to the next word. If it is not, the spell checker tries inserting, removing, substituting and swapping (or reversal of) letters to see if it can find any words from the dictionary. These four changes represent major error types. A limitation of this approach is that only words in the dictionary are considered correct, which may lead to both false negatives and false positives. Another limitation is that no account is taken for the surrounding words. Further, many spell checkers only check for errors at one place in the word (or at least this used to be the case, e.g. (Kernighan et al., 1990)).

4.1 Levenshtein distance

Levenshtein distance (LD), also called edit distance, is a measure of similarity between two strings. The distance is the number of deletions, insertions, or substitutions required to transform the source string, *s*, into the target string, *t*. For example, if *s* is ”thing” and *t* is ”thing”, then *LD(s,t) = 0*, because no transformations are needed. The strings are already identical. If *s* is ”thing” and *t* is ”think”, then *LD(s,t) = 1*, because one substitution (change ”g” to ”k”) is sufficient to transform *s* into *t*. A greater Levenshtein distance, means more different strings (Gilleland,).

The Levenshtein distance can be found by (Gilleland,):

1. Set n to be the length of s. Set m to be the length of t. If n = 0, return m and exit. If m = 0, return n and exit. Construct a matrix containing 0..m rows and 0..n columns.

2. Initialize the first row to 0..n. Initialize the first column to 0..m.

3. Examine each character of s (i from 1 to n).

4. Examine each character of t (j from 1 to m).

5. If s[i] equals t[j], the cost is 0. If s[i] doesn't equal t[j], the cost is 1.

6. Set cell d[i,j] of the matrix equal to the minimum of: a. The cell immediately above plus 1: d[i-1,j] + 1. b. The cell immediately to the left plus 1: d[i,j-1] + 1. c. The cell diagonally above and to the left plus the cost: d[i-1,j-1] + cost.

7. After the iteration steps (3, 4, 5, 6) are complete, the distance is found in cell d[n,m].

4.2 Noisy Channel Model

The noisy channel model of Shannon (1948) has been applied successfully to many different problems, spell checking among them. The models has two components: a source model and a channel model. In applying this to the production of natural language text, it is assumed that a person choses a word *w* to output, but that the noisy channel induces the person to output string *s* instead.

Kernighan et al. (1990) describe how probability scores for candidate corrections can be found using a noisy channel model. Using a Bayesian argument, the intended correction, *c*, can often be recovered from the typo, *t*, by finding the correction *c* that maximizes $Pr(c)\ Pr(t|c)$. The first factor, $Pr(c)$, is a prior model of word probabilities. $Pr(t|c)$ is a model of the noisy channel that accounts for spelling transformations on letter sequences, such as insertions, deletions, substitutions and reversals.

The first step of Kernighan et al. is proposing candidate corrections, which means finding words that differ from the typo t by a single insertion, deletion, substitution or reversal. These transformations are named from the point of view of the correction, not the typo. For example, for the typo *simpe*, could be the word *simple* transformed by a noisy channel by replacing the *l* with nothing at position 5.

When a list of candidates have been generated they are scored as described above. $Pr(c)$ is estimated as $(freq(c) + 0.5)/N$, where $freq(c)$ is the number of times that a word *c* appears in the training corpus and *N* is the number of words in that corpus. The conditional probabilities are computed from four confusion matrices: *del[x,y]*, the number of times that the characters *xy* (in the correct word) were written as *x* in the training set; *add[x,y]*, the number of times *x* was written as xy; *sub[x,y]*, the number of times that *y* was written as *x*; and *rev[x,y]*, the number of times that *xy* was written as *yx*. From these matrices probabilities are estimated by dividing by *chars[x,y]* or *chars[x]*, the number of times that *xy* and *x* appeared in the training set, respectively.

4.3 General issues for spell checkers

What size should the dictionary be? A larger one is of course preferred, but with many unusual words there is an increased risk that they will match a misspelled word. A larger dictionary takes longer to search, and on smaller devices storage might be an issue. Compiling a dictionary is not a trivial task, as possible text sources may contain errors and many proper nouns.

5 Implementation

This paper will describe the implementation of a spell checker that uses a noisy channel model, which allows for more than one deletion, insertion, substitution or reversal between the typo and the correction (Kernighan et al. only have one), and where the word chosen as the correction by the user updates the programs confusion matrices. The spell checker is written in java, and the interface is Mac OS X's cocoa.

5.1 What the user sees

The user is met with a text editor window and a spell checker window. When the ”check spelling” button (in Swedish: *kolla stavning*) is pressed, the first word that the spell checker considers to be misspelled is selected (highlighted) and the user is provided with five suggestions along with the misspelled word, in the spell checker window (see Figure 1). To correct a word, the user selects a word from the list and presses the ”correct” button (*rätta*). There are several other buttons in the spell checker window. Guess (*gissa*) makes new suggestions based on the word in the text box. This is useful if the user wants to make changes to the original misspelling and get new suggestions, if the intended word was not in the list. For more on this, see the description of the CHECK system in the conclusion. Ignore (ignorera) adds the selected word to a temporary list of words which will not be regarded as misspelled. Add word (*lägg till ord*) will add the selected word to the program's dictionary. Find next (*sök nästa*) will skip the selected word and move on to select the next misspelled word.

5.2 Program structure

The SpellChecker class handles all real spell checking, while the SpellInterface class handles the interaction with the interface. The other main classes are Corrector, Scorer, and Trainer. LevDistance calculates the Levenshtein distance and gives operations associated with this distance. There are several classes for holding information about words and corrections: error/correction types (Correction), a word that is a possible correction (CorrectionWord), and a typo and its possible corrections (Typo).

When the ”check spelling” button is pressed, for each word, SpellInterface asks SpellChecker whether or not it is misspelled. If SpellChecker finds that the word is not in the dictionary, it asks Corrector for a list of words which are possible corrections for the typo. Corrector first get a list of words which are within a certain range of the typo's length. It then uses LevDistance to calculate the edit distance between these words and the typo, and gets a list of operations necessary to transform the correctly spelled word into the typo. A CorrectionWord holds this word and the list of operations. SpellChecker then sends the list of CorrectionWords it receives from Corrector to Scorer asking for the n best ones. Scorer holds the confusion matrices and the character matrices described above in section 4.2. For each CorrectionWord, Scorer calculates a score, based on the Bayesian argument described in the same section. The probability of is taken to be the probability of the suggested word times the product of the conditional probabilities of the operations needed to transform the suggested word into the typo. The probability of the suggested word, *Pr(c)* was estimated as *(freq(c) + 0.5)/N* by Kernighan et al. (1990). Unfortunately, this prior model cannot yet be estimated for this program, as a corpus has not yet been gone through for this purpose. Scorer thus considers Pr(c) to be 1. This should be rectified. Scorer returns to SpellChecker a list of n CorrectionWords, sorted by score. If there are not five non-zero scores, words are added to the list according to edit distance. Finally, SpellChecker makes a Typo with the misspelled word, its list of all possible CorrectionWords and the list of suggestions that scorer provided. These suggestions are then displayed to the user.

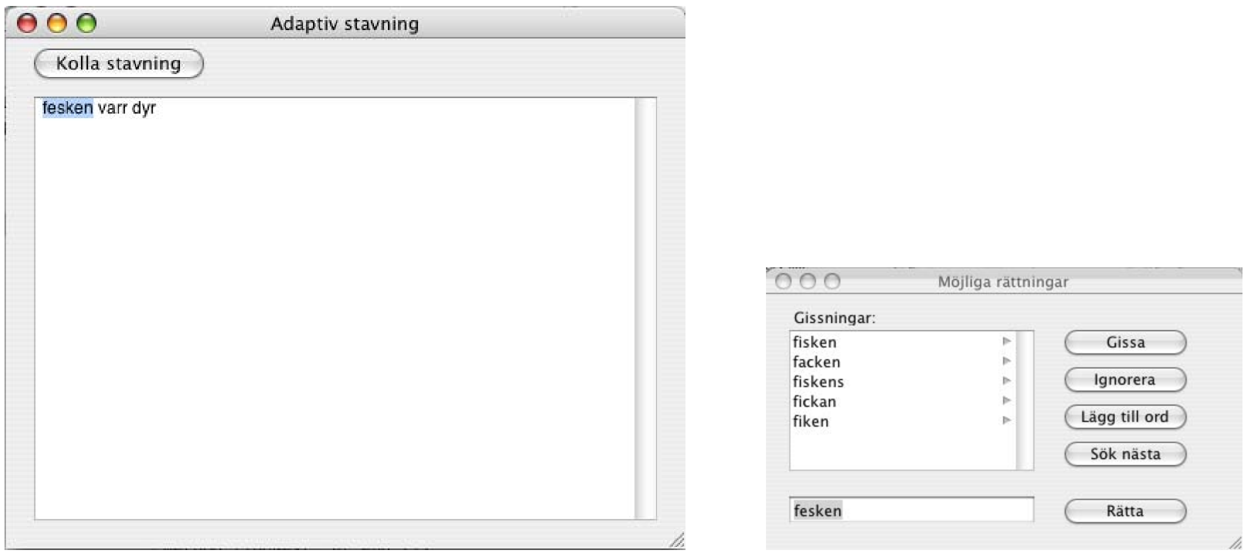
The Trainer class is used to initially fill and update the confusion and character matrices. For the initial training, a list of common misspellings in Swedish has been used. The matrices continue to be updated as the user uses the spelling checker. Feedback is sent to Trainer every time the user selects a word as a correction.

6 Evaluation

Unfortunately, this spell checker has yet to be tested on texts by dyslexics. Hopefully this will be done in the near future. A corpus for use in calculating prior probabilities and for enlarging the dictionary should also be incorporated.

For the initial training of the confusion matrices more training material is needed, since the program is of no use if these are not representative of common errors in spelling in Swedish. However, filling the matrices is not easily done, since texts or lists with both typos and corrections are needed, or it has to be done by hand. Further, it is difficult to obtain samples of dyslexic writing, since dyslexic people tend to write less and be less inclined to share their writing. Getting a lot of text from one person is even more difficult. And since it cannot be said that all dyslexics make the same mistakes, a set of matrices suitable to one dyslexic person might not be suitable for another.

There are some problems associated with updating the confusion matrices based on the users' chosen correction. One is that the user might have chosen the wrong correction, meaning that the matrices are not updated correctly. However, if this does not occur very often it should have little impact on the scores calculated from the matrices, given that the matrices are well filled. A larger quantity of mistakes might actually lead to more efficient training.



fesken varr dyr

fisken
facken
fiskens
fickan
fiken

fesken

Figure 1: The spell checker in action

The approach used here assumes that there is some consistency to an individual's spelling patterns. More research should be looked at to decide whether or not this is actually the case. Also, a major short coming in this implementation is that it only to some extent addresses phonetic misspellings. That is, only phonemes represented by one letter can be dealt with.

The Scorer always returns a list of n possible corrections. It might be better if it only returned those who had scored above zero. However, given the limited confusion matrices in this version of the program, this may lead to no suggestions being made. In a future version, a more sophisticated selection based on score and variations in scores might be used.

The dictionary is stored in java's hashtable. More efficient methods of storage might be preferred. For example Stava (Kann et al., 1998) uses Bloom filters. Other necessary improvements to make the spell checker useful is to incorporate inflections so all don't have to be in the dictionary, and to have some way to check compound words. Another desirable extension is some kind of context sensitivity.

This work will not address some important aspects of spell checkers for Swedish, such as how to deal with compound words and the inflection of verbs. Another issue which will not be addressed, but which is a problem in most languages, is that a word might be misspelled for the intended meaning while being in the dictionary as it is a correct word. For example, ”witch house?” would be viewed as correct by a spell checker even though the intended sentence was probably ”which house?”. These mistakes are difficult to discover without a grammatical analysis or a more direct listing of words which sound alike

7 Conclusion

The application of a simple user model for a spelling correction program based on a noisy channel model seems promising. While the spell checker is by no means complete, this rudimentary structure does surprisingly well. Of course, this said before the program has been put to a real test. The idea of using the confusion matrices as a rudimentary user model is promising, and is not limited to use for dyslexic people, though they might need an adaptive spell checker more. The existence of research on whether or not dyslexic people make different spelling mistakes should be looked into.

Ashton describes the CHECK strategy, developed to help students use spell checkers more effectively and independently. This strategy makes use of the ”change to” box that many spell checkers have and that most let the user type in. It should be explained to the student that they can make changes in the word in this box and then press the ”guess” button (or one with equivalent function) to generate a list of suggestions for the new word. If the new word is closer to the intended word, it might appear in the new list. This can be repeated as many times as is necessary, but it is best if only one type of change is made at a time. CHECK stands for

- Check the beginning sound of the word
- Hunt for the correct consonants
- Examine the vowels
- Changes in suggested word lists may give hints
- Keep repeating steps one through four

One possibility is to include instructions such as these in the program, if the user requires extra help.

Right now there are five words in the list of suggested words given to the user. If the intended word is not in that list, the user has to employ a method similar to the CHECK strategy. Ideally, the spell checker should find the target word in the first pass, but if it does not, a ”more suggestions” button, which adds five more suggestions to the list, might be useful.

The user model described by Spooner and Edwards (1997) is derived from a cognitive model of language production. More complex rules are used, which intend to describe permutations of errors typically made by dyslexic writers. The choices the user makes from the list of suggestions is used as feedback to measure and improve the user model’s accuracy. Spooner (1996) claims that an attempt to extend the methods of checking for the four classic errors in multiple combinations (i.e. allowing for an edit distance larger than one) would both take longer and produce more suggestions. This is exactly, though perhaps not in the most efficient way, what has been done in this implementation, without any problems with speed.

Brill and Moore (2000) offer improvements on the work of Kernighan (1990), by using a more generic error model of string-to-string edits and so modeling substitutions of up to 5-letter sequences (e.g. ent being mistyped as ant, ph as f, etc.) within the framework of a noisy channel model. They find that this handles phonetic errors better than previous methods. It does however make residual errors, many which have to do with word pronounciation. To address this, Toutanova and Moore (2002) build two different error models using the Brill and Moore algorithm, one letter-based one and one based on a phone-sequence-to-phone-sequence error model. Since the better correction phonetic errors is especially interesting for a spell checker for dyslexics, their methods should certainly be looked into.

Apart from more support from research on the kinds of spelling mistakes that dyslexics make, getting a program that handles phonetic errors better is vital. Having more filled confusion matrices is also very desirable, so that one might have several sets that can serve as the beginning for different user groups, such as ”regular users”, dyslexics, or people who speak Swedish as a second language, where the spell checker could be adjusted after mother tongue.

Acknowledgements

Thanks to Pierre Nugues for all advice in developing this project. Also, I would like to thank Åsa Wengelin for getting me interested in the subject of spell checkers to begin with.

References

Tamarah Ashton. Making technology work in the inclusive classroom: A spell checking strategy for students with learning disabilities.

Eric Brill and Robert C. Moore. 2000. An improved error model for noisy channel spelling correction. *Proceedings of ACL-2000*.

Michael Gilleland. Levenshtein distance, in three flavors.

T. Hoien and I. Lundberg. 1999. *Dyslexi från teori till praktik*. Natur och Kultur.

Viggo Kann, Rickard Domeij, Joachim Hollman, and Mikael Tillenius. 1998. Implementation aspects and applications of a spelling correction algorithm.

Mark D. Kernighan, Kenneth W. Church, and William A. Gale. 1990. A spelling correction program based on a noisy channel model. *Proceedings of the Thirteenth International Conference on Computational Linguistics*, pages 205–210.

Alfred Kobsa. 2001. Generic user modeling systems. *User Modeling and User-Adapted Interaction*, 11:49–63.

Claude Shannon. 1948. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423.

Roger I. W. Spooner and Alistair D. N. Edwards. 1997. User modelling for error recovery: A spelling checker for dyslexic users. *User Modeling: Proceedings of the Sixth International Conference, UM97*, pages 147–158.

Roger Spooner. 1996. Dphil thesis proposal a computerised writing aid for dyslexic people.

Kristina Toutanova and Robert C. Moore. 2002. Pronounciation modeling for improved spelling correction. *Proceedings of the 40th Meeting of the Association for Computational Linguistics(ACL-2002)*, pages 144–151.

Maria Virvou and Katerina Kabassi. 2002. Improving agent control for user modeling. *First International IEEE Symposium on Intelligent Systems*, pages 73–78.

Geoffrey I. Webb, Michael J. Pazzani, and Daniel Billsus. 2001. Machine learning for user modeling. *User Modeling and User-Adapted Interaction*, 11(1-2):19–29.

Automatic learning of discourse relations in Swedish

Stefan Karlsson
Department of Computer Science
Lund University
f98sk@efd.lth.se

Abstract

This report describes some experiments with a statistical technique that extracts discourse relations in Swedish. Three types of relations are used: Cause-Explanation-Evidence (CEV), Contrast and Elaboration. The method is evaluated by building two-way classifiers, with the results: Contrast vs. CEV 64.5%, Contrast vs. Elaboration 56.6% and CEV vs. Elaboration 54.9%. The conclusion is that the technique, with improvements or modifications, seems to be usable to extract discourse relation in Swedish, but further investigations are necessary.

1 Credits

Most of the corpora used in this project have been provided by Lars Aronsson from the Runeberg project. As a tool to identify nouns and verbs the Granska grammatical tool was used, which was provided by Jonas Sjöbergh at KTH. Finally the Stockholm-Umeå corpus was needed to build the Granska tool, and was provided by Sofia Gustavfson-Capkova also at KTH.

2 Introduction

There are relations in discourse, for example “I did put my coat on this morning, because it was cold”. The second clause of the sentence is describing a cause of the fact stated in the first clause. Another example is the contrast: “The car is green on one side, but red on the other”. No good techniques have yet been developed to automatically identify different types of discourse relations. A useful application would perhaps be to extract all causes from discourses and put them into a knowledge base. For example you could ask: “What causes crops to grow?”, and a knowledge base agent would answer “the sun”, “the earth” and “rain”.

In many cases there are markers that indicate a particular discourse relation, as in the examples above “because” and “but”. However you could also say for example “The car is green on one side, and red on the other”. A human might conclude that the car being red on one side is in contrast to the car being green on the other side, but since it’s not explicitly marked, some more elaborate reasoning strategy is needed.

This project work was an attempt to implement a algorithm that makes a choice whether two word spans in Swedish can be classified as together constituting a particular discourse relation. The algorithm has been developed by Marku and Echihabi (2002), and is here implemented for and tested on Swedish discourse.

3 A statistical model

The approach taken by Marcu and Echihabi (2002) was to build a simplistic statistical model. Basically there might be some word pairs that are frequent in contrasts, for example “green” and “red” as in the example above, and other pair in explanations, i.e. “cold” and “coat”. To capture these patterns a table can be made, where the word pairs formed from each different combination of words from the first and second part are counted. This is the cartesian product of the text spans W_1 and W_2, defined as $(w_i, w_j) \in W_1 \times W_2$. The table becomes non-commutative since the first text span, so to speak, ends up in the columns of the table and the second text span ends up in the rows.

The probability that two text spans forms a particular relation can be calculated as follows:

$$P(r_k|W_1, W_2) = \frac{P(W_1, W_1|r_k)P(r_k)}{P(W_1, W_1)}. \quad (1)$$

The factor $P(W_1, W_1|r_k)$ can be estimated with $\prod P((w_i, w_j)|r_k)$, were w_i and w_j symbolizes the words in each span. $P((w_i, w_j)|r_k)$ is directly calculated from the table.

Of course not all possible word pairs, that might be encountered in for example contrasts, will be counted in the table, which makes it necessary to shift some probability mass to these previously unseen pairs. Marcu and Echihabi used the Laplace method.

4 Extraction of sentences

First of all three discourse relations were used in the experiment. These are Cause-Explanation-Evidence (CEV), Contrast and Elaboration. [1]

To be able to experiment with the technique, Swedish corpora were attained from the Runeberg project (45 million words) and the European Parliament (16 million words). The first difficulty was to find good Swedish markers for extraction of training examples. By inspection of sentences the extraction patterns in table 1 were judged to be good enough. Since a great portion of the corpora was from Nordisk Familjebok from the end of the 19th century and the beginning of the 20th century, some old markers were used (“ty” and “ehuru”), together with some more modern ones (“eftersom” and “trots att”).

The corpus was divided into a training set (99.5%) and a test set (0.5%). Results from Marcu and Echihai (2002) indicate that using only nouns and verbs makes a steeper training curve, and since quite a small corpus was used this approach was taken. For this purpose the Granska grammatical tool [2] was used. All non-nouns and non-verbs were identified and marked, and later discarded in all experiments. The Granska tool also helped to identify sentences in the corpora.

Finally the training examples were extracted with 156762 Contrasts, 43159 CEV and 21072 Elaborations, and the testing set with 771 Contrasts, 176 CEV and 79 Elaborations.

[1] For a description of these, see Marku and Echihabi (2002) and literature referenced to from there.

Contrast
[BOS ...][, men ... EOS]
[BOS ...][, ehuru ... EOS]
[BOS ...][, fastän ... EOS]
[BOS ...][, trots att ... EOS]
Cause-Explanation-Evidence
[BOS ...][, därför att ... EOS]
[BOS ...][, eftersom ... EOS]
[BOS ... EOS][BOS Alltså ... EOS]
[BOS ...][, alltså ... EOS]
[BOS ... EOS][BOS Således ... EOS]
[BOS ...][, således ... EOS]
[BOS ... EOS][BOS Sålunda ... EOS]
[BOS ...][, sålunda ... EOS]
[BOS ...][, ty ... EOS]
[BOS ... EOS][BOS Ty ... EOS]
[BOS ... EOS][BOS Därför ... EOS]
Elaboration
[BOS ...][vilket ... EOS]

Table 1: Swedish extraction patterns used in the experiments.

5 Experiments

5.1 Methods of evaluation

To evaluate the technique, two-way classifiers were built to distinguish Contrast vs. CEV, Contrast vs. Elaboration and CEV vs. Elaboration. A decision is made by taking the maximum of $P(r_k|W_1, W_2)$ for each relation, where $P(r_k|W_1, W_2)$ is calculated from the table. In

[2] see www.nada.kth.se/theory/projects/granska/

equation 1, $P(W_1, W2)$ can be discarded, since it's the same for all relations.

An approach that would eliminate the factor $P(r_k)$, would be to put the same amount of sentences from each type of relation in the test set. However, since there were as few as 71 Elaborations in the test set, and as many as 771 Contrasts, this was instead emulated by taking the mean of the amount of correctly classified sentences from each relation. In this way the result is more statistically validated. So, for example in the case of Contrast vs. CEV, all contrast sentences from the test set were classified and the percentage of correctly classified sentences was calculated. The same thing was done for CEV sentences and finally the mean of these percentages taken as the result. This is similar to having the same proportions of contrasts and CEVs, meaning that $P(r_{contrast}) = P(r_{CEV})$ and this factor can be discarded.

During early experiments it was discovered that the Laplace method seemed to shift too much mass of probability to unseen word pairs. In one table there were 900 times as many zeros as actual counts in the table. Therefore Lidstone's rule was used instead, which amounts to setting:

$$P((w_1, w_2)|r_k) = \frac{(count + \lambda)}{(total + \lambda \cdot cardinal)}, \quad (2)$$

where cardinal is the number of entries in the table. It was found that a lambda of 0.05 seemed to maximize the accuracy of the classifiers; a value that was kept during all subsequent experiments.

5.2 Results and an improvement

The accuracy of the classifiers are presented in table 2. A maximum result of 64.5% in the Contrast vs. CEV condition is in the same realm as the results from Marcu and Echihai (2002), who had between 60% and 70% in most conditions.The results for Elaboration were worse, with 57% in the Contrast vs. Elaboration case.

During these experiments one thing wasn't considered however. The method so far has been consisting of using a left and a right part in the training examples. A problem with this can be illustrated with the two sentences "I put my coat on, because it is cold" and "It is cold, thus I put my coat on". Both are causal and basically state the same thing, but the order of the clauses are reversed. In the first case the training procedure would take "I put my raincoat on" as the first clause of the sentence and "it is cold" as the second, but in the opposite order in the second case. Perhaps it would be better to identify the cause and the effect in each sentence and train the table with these instead of using the left and right part of the sentences. To test this hypothesis the training with CEV relations was done again, but with the word spans put in logical order, that is causes to the left and effects to the right. The result was that 65.1% of the sentences were correctly classified in the contrast vs. CEV case.

	Contrast	Elaboration
CEV	64.5 %	54.9%
Contrast		56.6%

Table 2: The result of each evaluation.

5.3 A source of bias

Since texts from different time-spans were used, i.e. Nordisk Familjebok and Svenska Familj-journalen from the end of the 19th and beginning of 20th century, and modern discourse from the European Parliament, the results above might have been biased. At worse the classifiers only differentiates between old and new sentences, and not at all between different kinds of discourse relations. The problem is clear since in older Swedish, as in Nordisk Familjebok, "v" is often spelled "f" or "fv". For example "av" becomes "af" and "silver" becomes "silfver". Other differences might also influence the classifiers. It was actually found that using the contrast vs. CEV classifier on old vs. new sentences gave a result of 77%.

In an attempt to get around this problem, the tests sets were further divided. Sentences from Nordisk Familjebok and Svenska Familj-journalen were separated from sentences from the European Parliament, and old and new sets were thus formed. The CEV vs. contrast classifier was evaluated for each case. The results were 60% correctly classified sentences from the old set and 58% from the new.

6 Conclusions

Results around 60% clearly indicated that the classifier is better than random assignment of text spans to each class. Of course the accuracy is not high, but since marker words themselves are not used to train the classifiers, the accuracy can probably be increased tremendously. The corpora used were also quite small, and more training examples are needed to increase accuracy.

The results with elaboration are not very good, which first of all can be accounted to the fact that only 21072 training examples were used. Also, as Marcu and Echihabi (2002) states, there is some dispute regarding the existence of a well formed category of elaboration as a discourse relation. Perhaps this can be used as a way to experimentally find evidence for well formed categories.

The fact that two quite different types of discourse was used, that is old and new, had a great impact. Since the Contrast vs. CEV classifier was better at discriminating between old and new sentences than Contrasts and CEVs, the bias introduced was quite large. A conclusion might be that the method works better on restricted discourses. For example a classifier trained on technical reports would be very good at identifying discourse relations in other technical reports.

There are some simple improvements that could be made. For example, since there seems to be no intrinsic order in contrast relations the table should be made commutative, that is training both "forward" and "backward". This was however not judged as a critical point, since there were more than 150000 training examples of contrasts. Another thing is to find the value of lambda in Lidstones rule, that maximized the accuracy. Using the value of 0.05, instead of 1 as in the Laplace method, greatly improved the classifiers, and more improvement can probably be made.

To sum up, some evidence is presented that this constitutes a feasible technique for automatic extraction of discourse relations in Swedish, at least with some improvements, but further investigations would be necessary to accurately evaluate this contention.

References

Daniel Marcu and Abdessamad Echihabi. 2002. An Unsupervised Approach to Recognizing Discourse Relations. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics ACL-2002* (available at www.isi.edu/m̃arcu/papers/relations-acl2002.pdf), Philadelphia, PA, July 7-12

A writing assistant using language models derived from the Web

Sven Karlsson
Department of Computer Science
Lund University
sk@svenk.nu

Abstract

The web is the largest amount of text ever available to man, and the search engines has classified and counted words on a large portion of it. This give us access to huge copra in a number of languages. This article tries to se how the different measurement methods behave and what possible use they might have. The measurements used in this article are uni/bigrams, bi/trigrams, mutual information (Church & Hanks 1990), T-scores (Church & Mercer 1993), and log likelihood (Dunning 1993). A lot of work has been done to make a visual display that makes it easy to compare the differ measurements

1 Credits

This project is a part of the course DAT171 Language Processing and Computational Linguistics at Lund University. The idea of using the web as a source for statistics came from senior lecturer Pierre Nugues.

2 Introduction

Language was first grammar came second. The set that is described by grammar is not the same set as the modern language. These are the key points why I think statistic analyzing and the brute force approach has been so successful. Though I could not fully comprehend Wittgenstein's language theory I sympathizes with the basic idea that language is a game governed by a set of commonly agreed set of rules. The rules change over time and between groups. Think of football, American football, rugby, Wales football and football originate from the same set of rules and they have their likeness but being good at one of the those not mean that you are good at all of them. Now that we have access to huge amounts of corpora we need find ways to use it. We have looked at uni/bigrams, bi/trigrams, mutual information, T-scores, and log likelihood as measurements for word collocation. The Google API didn't give us any possibility to use a window for the words so had to make do with regular bi & trigrams. There are of course other statistical test we could have used such as t-scores and $\chi 2$. The criticism about t-scores has been that is assume normal distribution and is therefore not suitable for limited copra, so would been a candidate but time did not allow that.

3 Background

Being dyslectic, I produce a tremendous amount of errors and I haven't found a speller that can find them all. My dream is to make a speller that would catch more errors and help you before you make them. Today I usually let someone proofread what I write, but there are four problems with this:

1. I'm running out of friends because it can be quite time consuming.
2. The proofreader needs domain specific knowledge.
3. Some persons are better proofreaders than others.

4. I need to understand and be able to work together with proofreader.

So if I hand out this paper to ten people and ask them and to proofread and correct it for me we would probably end up with ten different documents. Every person has is view of language should look like and all of them would or could be grammatically correct. Grammar alone we will not catch awkward expirations or faulty implications, so we I need tools that can handle this.

4 Practical uses

I have mentioned spellers before. I think it would work great as a complement finding words that have been misspelled in a way that they are correctly spelled words but not the intended word. I also think it will get the suggestions for the replacement word up to a level of a human user. It can also pick up problems with sentencing that is missed by the normal grammar checker, but it won't be able to give any clues to what is wrong in this case. In Swedish, there is a special problem with split compounds that probably also could be solved. There are more areas than spelling and proofreading that will benefit from this. To simplify languages is a big market for those who write manuals and technical documents or other literature where it is important that the reader can understand. It does also make things easier when you want to translate a text, especially if it will be machine translated. It could also help people who write in a non-native language. Here I think we can go a step further and create a tool that will be able to suggest word and actually help the writer in an early stage. There are products like these on the market Co:Writer® by Don Johnston inc. This program is for kids and uses very restricted topic dictionaries but I think that there will come much more advanced tools in a near future.

5 The statistic measurements

The five methods compared in this paper range from very simple to complex. The common feature of the methods is the use of the count of uni, bi and trigrams. In all the methods we used the scores and the resulting ranking and discarded level of significance.

5.1 Uni/Bigram and Bi/Trigram

These are the two simplest measurements methods. Here we look at the quotient between unigram and bigram or bigram and trigram. The idea is to counter the effect of commonly used words by looking at the quotient, which means if a certain combination of word is likely it's also likely that a smaller part of it also will be likely. The problem here is that we still do not compensate for extremes in occurrence. But still is an easy and fast way that gives very interesting results for some problems. The big difference between Uni/Bi- and Bi/Tri-grams and the other methods are that they are merely balanced counts as opposed to the other methods, which are based on significance.

5.2 Mutual Information

Fano (1961: 27–28) defined mutual information between particular events x and y, in our case the occurrence of particular words. Pointwise Mutual Information was presented by Church & Hanks in 1990 and is the method we used. Mutual Information measurements are better at compensating for rare word then simple Uni/Bigram and Bi/Trigram. This mean that we find rare word that when used their used together. When Church & Hanks wrote their article they saw a use for tool for lexicographers so they automate the processing of copra. The formula we used looks like this:

$$I\left(w^1, w^2\right) = \log_2 \frac{NC\left(w^1, w^2\right)}{C\left(w^1\right)C\left(w^2\right)}$$

5.3 T-Scores

T-Scores is standard statistical test that looks at the difference between the observed and expected means. T-Scores according to Church & Mercer 1993 better picking out grammatical patterns or combinations of very frequent word such as "and of" or "and the". The formula looks like this

$$T\left(w^1, w^2\right) = \log_2 \frac{C\left(w^1, w^2\right) - \frac{1}{N} C\left(w^1\right)C\left(w^2\right)}{\sqrt{C\left(w^1, w^2\right)}}$$

5.4 Log likelihood

Log likelihood (Ted Dunning 1993) is much more complex than the other measurements we used and claims that it is more accurate seen from a statistical viewpoint. The main advantage with this method is that it can be used on small corpora that are not binomially distributed. In our case we can assume that we have a binomially distributed corpora as we have huge or might I say ridiculously large corpora.

6 Implementation

The implementation consists of tree parts the web connection (Google API), calculations and GUI/displaying (see Appendix A). As the web transactions take a lot of time, every look up take up to 3 sec. and the average length of a sentence is ten words and as we access it 3 time once for unigrams, bigrams, and trigrams so it can take up to 2 minutes. At times when load is heavy, it can time out and fail a search completely and there is not much to do about the time it takes to do a lookup.

6.1 Google API

Google provides a SOAP[1] API with example code in Java. The API makes it possible to access the Google search engine from your program and gives an easy way to retrieve the result. A search via the API should give you the same result as normal web search, but it doesn't as we can se from Table 1 & 2. If you don't use language restriction as in Table 1 the errors are so grave that the result must be impaired.

Word	Google	api	%
The	3670000000	failed	
Of	2440000000	1530000000	0,627049
And	2390000000	1510000000	0,631799
To	2320000000	1460000000	0,62931
A	2240000000	1410000000	0,629464
You	213000000	426000000	2
Was	20100000	245000000	12,18905
An	16000000	300000000	18,75

Table 1. Google & api no restriction.

Word	Google	api	%
The	18300000	12700000	0,693989
Of	16700000	11600000	0,694611
And	17400000	12100000	0,695402
To	17200000	12100000	0,703488
A	17800000	12400000	0,696629
You	16100000	11200000	0,695652
Was	16200000	11300000	0,697531
An	15800000	11300000	0,71519

Table 2. Google & api language restriction.

What causes these results is unclear. One theory is that it is different languages or control characters that could cause these problems. This may explain some of the grave errors in Table 1 but it doesn't explain the 30% difference between a normal Google search and an API search. I restricted the search to include only English and only the body text in Table 2. A difference that remains and can cause problems is the three percent variation in lookup between words. **And** and **To** has a difference in excess to 200000 when we use a restricted Google search, but when we use the API we get the exact same answer. There is a possibility to check if number of hits is an estimate or if it's an exact value, but when we are dealing with English almost everything is an estimate so it is of no greater use.

When restricting a search, you choose from 28 different languages and 240 domains. It is also possible to restrict a search to just text or links. To even further restrict a search you can put on date restriction or search just one site.

6.2 Scaling

To get the scale right was a time consuming job. And as discussed before we are only interested in the scores and the resulting ranking.
The scaling model can be set in different ways linear, logarithmical or preset scale; they can either be self-adjusting or have fix min and max. Each problem needs is own scaling so that result really shines through the noise. It would also be good if the users could make their own adjustments.

[1] SOAP, or the Simple Object Access Protocol, is an XML based protocol for accessing remote objects over the network.

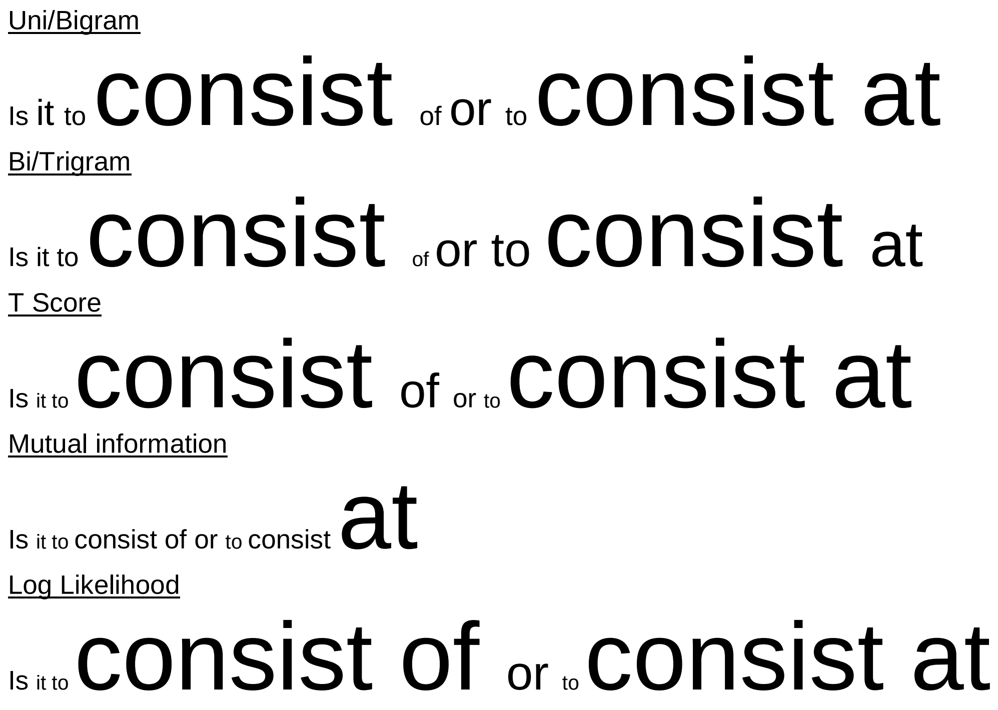


Figure 1. consist at consist of.

7 Results

We start with a favorite example *Will it consist of or will it consist at*.

Here we can see (Figure 1.) the expected difference between **of** and **at** in all 5 of the measurements. We can also se that the word *consist* scores as very unlikely alternative except with the mutual information. This could be explained by the fact that *consist* is a rare word compared to the rest of the words. The behavior that mutual information shows is exactly what we want. A divergence that is harder to explain is the result from log likelihood. Any idea mail me!

In Table 3, we can see the scores from the Goggle API. *Consist* sticks out and is a tenth as common as any of the other words are.

Unigram count	
Is	11600000
it	10900000
to	11100000
consist	1200000
of	11600000
or	11000000
at	10500000

Table 3. Unigrams.

If we look at the bigrams, we find that even without any calculation it is obvious that **consist of** is the correct form. We can also see that all the bigrams that contain *consist* are much lower than the others.

Bigram count	
Is it	5880000
it to	7110000
to consist	123000
consist of	976000
of or	3440000
or to	6790000
to consist	123000
consist at	2130

Table 4. Bigrams

When we look at the trigrams it is not so obvious **to consist at** is wrong because we have **or to consist** that scores equally low. But what we can see is that the proper way to use **consist** in is the phrase **to consist of.**

Trigram count	
Is it to	318000
it to consist	1090
to consist of	107000
consist of or	5660
of or to	32800
or to consist	192
to consist at	299

Table 5. Trigrams.

Word	T-Score	Mutual	Log Like
it	0,506897	2424,219	11,86122
to	0,652294	2665,84	12,07437
consist	0,010165	350,2004	9,416546
of	0,820168	987,7525	12,46565
or	0,296552	1853,864	11,07464
to	0,617273	2605,124	11,99476
consist	0,010165	350,2004	9,416546
at	0,00179	42,76772	3,769503

Table 6. T-scores Mutual information Log Likelihood

This is for the person who wants to go to the motions and see that I did the math correctly.

8 Further work

- Finding more good examples to try out.
- Tweak the scales and make it to do simple math operation between the ranks of the different measurements.
- Test it other languages and specific domains.
- Implement methods for smoothing N-grams for sparse data.
- Test other measurements like Z-score and χ^2.
- Building a large local database of uni, bi and trigram would be the first step this would cut execution drastically time and make evaluations of methods, measurements and problem a more reasonable task.
- See if it is possible to make word prediction

9 Conclusions

The conclusion must be that it is possible to use the web as base for collecting language data. There seem to be great possibilities to make a lot of different tools out of the data available and the amount of data makes it easier and produces more reliable data. The sparse testing that we have done so far shows good promises for the future.

References

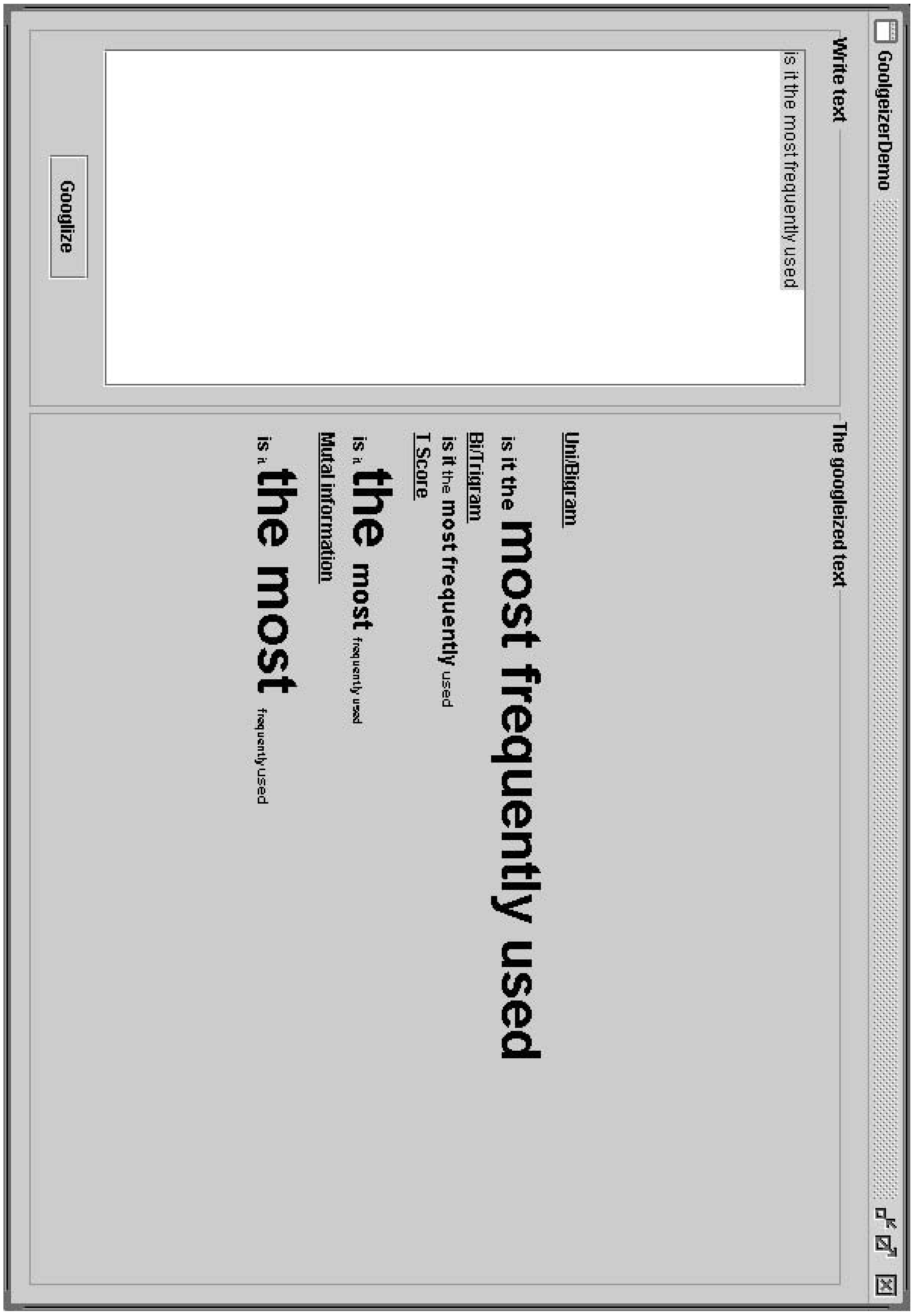
Ted E. Dunning 1993 *Accurate Methods for the Statistics of Surprise and Coincidence*. Computational Linguistics : Vol. 19, No. 1, pp.61-74

Christopher D. Manning and Hinrich Schütze, 2000. *Foundations of Statistical Natural Language Processing.* , MIT Press, Cambridge Mass., 1999.

Ken. W. Church and P. Hanks. 1990. *Word Association Norms, Mutual Information, and Lexicography*, Computational Linguistics, Vol. 16, No. 1, pp. 22-29

Ken. W. Church and R. Mercer. 1993. Introduction to the special issue on Computational Linguistics using large corpora. *Computational Linguistics*, 19 (1): 1--24.

Appendix A. My program



is it the most frequently used

Uni/Bigram

is it the most frequently used

Bi/Trigram

is it the most frequently used

T Score

is it the most frequently used

Mutal information

is it the most frequently used

Automatisierte metrische Analyse lateinischer Dichtung

Ulrich Klauer
Lunds universitet/
Georg-August-Universität Göttingen
`ulrich.klauer@stud.uni-goettingen.de`

Abstract

This paper describes a prototype system for the automated metrical analysis of classical Latin verse. Even though the Latin writing system not at all guarantees a perfect rendering of the features relevant for the metric in a verse, it is quite well possible to at least narrow down the results to a few candidate solutions. Use of a dictionary should greatly improve the performance and deliver a unique solution most of the time.

1 Einführung

Dichtung zeichnet sich dadurch gegenüber Prosa aus, daß in ihr der sprachliche Ausdruck durch bestimmte Regeln eingeschränkt ist. Sie können sich etwa auf die zulässige Abfolge von betonten und unbetonten Silben beziehen, wie in den Versformen der meisten modernen europäischen Sprachen, oder auf die Kombination bestimmten Wörter, wie etwa im Reim; daher bezeichnet man die Sprachform in Gedichten auch als „gebunden“.

In der lateinischen Dichtung der Antike bezogen sich solche dichterischen Formen überwiegend auf die Abfolge unterschiedlich langer Silben. (Wir bezeichnen heute lange Silben als schwer, kurze als leicht; das vermeidet Verwirrung, weil schwere Silben sowohl lange als auch kurze Vokale enthalten können, leichte Silben allerdings nur kurze.) Diese Form der Versbildung heißt quantitative Metrik. Ein zulässiges Muster für einen Vers kann etwa folgendermaßen aussehen: –⏔–⏔–⏔–⏔–⏔–×. Dabei steht – für eine schwere, ⏑ für eine leichte und × für eine beliebige Silbe; ⏔ steht für wahlweise eine schwere oder zwei leichte Silben. Dieses Versmaß heißt (daktylischer) Hexameter.

Wie auch in den meisten modernen Sprachen ist die lateinische Orthographie alles andere als vollkommen: Lange und kurze Vokale werden mit denselben Buchstaben wiedergegeben, sind also nicht zu unterscheiden; Diphthonge werden mit zwei Buchstaben geschrieben, die aber auch für getrennte Laute stehen können (etwa *a-e*); das Zeichen *i* steht sowohl für den Vokal *i* als auch den Konsonanten *j*. Dazu kommen zahlreiche weitere Variationsmöglichkeiten etwa in der Silbenbildung, die sich in der Schrift nicht widerspiegeln.

Trotzdem ist es erfahrungsgemäß für Menschen relativ einfach möglich, auch unbekannte Verse beim ersten Lesen überwiegend korrekt zu skandieren, also metrisch zu analysieren. Dies gilt selbst dann, wenn viele Vokabeln unbekannt sind und also keine Informationen aus Vorwissen ergänzt werden können. Es sollte daher möglich sein, auch einen Computer mit Heuristiken zu versehen, wie Menschen sie in dieser Situation anwenden, und ihn damit Verse automatisch analysieren zu lassen.

2 Umsetzung

Für die Realisierung des Projekts wurde Haskell als Programmiersprache gewählt; seine algebraischen Datentypen und seine bequeme Speicherverwaltung machen es einfach, auch komplexere Datenstrukturen zu realisieren.

Das Programm betrachtet jeweils nur einzelne Verse. Die Analyse erfolgt in fünf Schritten; jede Verarbeitungsstufe erhält nur die Ausgabe des vorangegangenen Schritts. Gibt es in einem Schritt mehrere Interpretationsmöglichkeiten – etwa *ae* als Diphthong oder als zwei getrennte Vokale –, so werden sämtliche Möglichkeiten betrachtet und an die nächste Stufe weitergegeben. Weniger wahrscheinliche Möglichkeiten, in diesem Fall die Interpretation als zwei getrennte Vokale, werden dabei mit „Strafpunkten“ belegt. Dies erlaubt es, äußerst unrealistische Interpretationen später auszufiltern.

1. Zunächst wird jedes Wort einzeln in Laute zerlegt; da ein Laut bei weitem nicht immer einem Buchstaben entspricht, vereinfacht dies die weitere Verarbeitung. Zugleich wird für jeden Laut vermerkt, ob es sich um einen Vokal bzw. Diphthong oder eine von verschiedenen Konsonantengruppen handelt. In diesem Schritt wird nicht versucht, zwischen langen und kurzen Vokalen zu unterscheiden: Da ein Vokalbuchstabe gleichermaßen für einen langen wie einen kurzen Vokal stehen kann, würde sonst nur nutzlos die doppelte Zahl von Lösungskandidaten erzeugt werden. Für die Zerlegung wird gewöhnliches Parsing mit rekursivem Abstieg verwendet.

2. Anschließend werden aus den Lauten Silben gebildet. Die lateinischen Silbenbildungsregeln sind relativ einfach: Von mehreren Konsonanten, die zwischen zwei Vokalen stehen, gehören alle bis auf den letzten zur ersten Silbe; gibt es nur einen Konsonanten, gehört dieser dagegen zur zweiten Silbe. Eine Kombination aus einem Plosiv und bestimmten Dauerlauten, etwa *tr*, wird dabei meist wie ein einzelner Konsonant betrachtet (sog. „muta cum liquida“). Auch für diesen Schritt reicht ein Parser mit rekursivem Abstieg, mit etwas Nachbearbeitung, aus.

3. Nun werden die bisher isoliert betrachteten Wörter wieder zusammengefaßt. Im allgemeinen genügt es, sie einfach zu verketten; allerdings treten an den Kontaktstellen der Wörter Konsonanten zum folgenden Wort über, wenn es mit einem Vokal beginnt. Treffen dort zwei Vokale unmittelbar aufeinander, so verschwindet zudem meist eine der Silben, es können aber ausnahmsweise auch beide erhalten bleiben (*Hiat*). Dies wird beim Zusammenfügen beachtet.

4. Nachdem jetzt die Silben im Vers vollständig bekannt sind, wird in diesem Schritt versucht, ihr Gewicht zu bestimmen. Silben, die einen Diphthong enthalten oder auf einen Konsonanten enden, sind stets schwer; Silben, auf deren Vokal ohne dazwischen stehende Konsonanten der Vokal der nächsten Silbe folgt, sind meist leicht. Bei den verbleibenden Silben hängt das Gewicht ausschließlich von der Länge des Vokals ab, die ohne Wörterbuch nicht bekannt ist, und kann daher nicht bestimmt werden.

5. Abschließend wird durch *pattern matching* gegen die in Frage kommenden Versmaße geprüft, welche Lösungskandidaten eines der Versmaße realisieren könnten. Dabei werden viele Kandidaten verworfen; so kann etwa die Silbenfolge schwer–leicht–schwer in einem Hexameter nicht auftreten.

Das Ausfiltern unrealistischer Kandidaten ist am besten nach Schritt 3 möglich, wenn die Strafpunkte für den gesamten Vers vorliegen.

3 Ergebnisse

Das Programm wurde mit rund hundert Versen aus der „Aeneis“ von Vergil (durchgehend Hexameter) und den „Amores“ von Ovid (abwechselnd Hexameter und Pentameter) erprobt, die während der Entwicklung des Programms nicht getestet worden waren.

Nur bei einem Vers gelang keine metrische Analyse; die korrekte Interpretation hatte so viele Strafpunkte, daß sie als zu unwahrscheinlich verworfen wurde. In den übrigen Fällen wurde jeweils die korrekte Analyse gefunden. Allerdings konnten dabei nur in rund einem Achtel der Fälle alle anderen Lösungskandidaten ausgeschlossen werden; in den übrigen Fällen blieben mehrere Möglichkeiten offen, so daß die korrekte nicht identifiziert werden kann. In einzelnen Fällen blieben zwar bis zu vierzig falsche Kandidaten übrig, doch ist dies die Ausnahme: Immerhin bei der Hälfte aller Verse waren es nicht mehr als zwei.

Eine deutliche Verbesserung dürfte der Einsatz eines Wörterbuchs bringen, in dem Vokallängen und Diphthonge verzeichnet sind. Obwohl auch dann noch nicht alle Mehrdeutigkeiten beseitigt sind – etwa bei Wörtern wie *malum*, die mit kurzem und langem *a* vorkommen, oder dem Hiat –, dürfte schon ein weniger umfangreicher Grundwortschatz die Analyse beinahe perfekt machen.

Will man weiterhin auf ein sicher aufwendig anzulegendes Wörterbuch verzichten, dürften sich die Verbesserungsmöglichkeiten dagegen auf kleinere Änderungen am Strafpunktesystem beschränken. Möglicherweise ist es aber noch möglich, bessere Heuristiken für die Deutung von *i* als *i* oder *j* zu finden: In seiner jetzigen Form hält das Programm häufig noch einen Vokal für möglich, wo ein menschlicher Leser intuitiv längst abgewinkt hätte.

4 Dank

Für das Programm habe ich die Parser-Infrastruktur adaptiert, die Lennart Andersson (Lund) in seinem Papier „Parsing with Haskell“ beschreibt. – Die Anregung, lateinische Metrik mit EDV zu behandeln, verdanke ich Marcus Deufert (jetzt Leipzig).

A text critiquing system for Swedish-speaking students of French

Fabian Kostadinov
Department of Informatics
University of Zürich
fkostadinov@gmx.ch

Jonas Thulin
Department of Computer Science
Lund University
jonasthulin@hotmail.com

Abstract

Making objective, quantitative linguistic analyses is a time-consuming and demanding task. Therefore, we have developed a language analysis system, which can be used for counting occurences of any given pattern. Rule trees and external dictionaries are supported. The program is written in 100 % Java 2 and Swing. Its strength is that rules can be easily added or modified and the main weakness is that it cannot assess language as well as a human can, since only quantitative criteria, and not semantic ones, are taken into account.

1 Introduction

Ever since people have been learning new languages besides their mother tongue, it ha been a question of interest of how to approach the target most efficiently. A number of different approaches and recommendations has been developped by linguists for support purposes. Interested in how to define reliable measures, which empower any teacher to state the language level of his pupils, some linguists over time developped such measures. They found out that the active and passive ability of speaking a language by a student, is reflected astonishingly well by certain grammatical indicators (besides, of course, the richness and variety of the student's vocabulary). So they generated rule sheets that can be applied to a text written by a student and return an indicator number, stating the student's level. Rules may include for example…

- how many times certain advanced verb tenses are being used throughout the text,
- how often the same substantial grammatical mistakes, as using a pronoun followed by a wrong verb form, are repeated,
- the average sentence length,

and much more.

However, the task of manually applying these rules to a text is time consuming and requires a profound understanding of grammatics, inflections and text analysis in general. As in today's world where one teacher often has to supervise several dozen students at the same time, such analysis cannot be done regularly; it is simply too demanding in terms of time and knowledge.

With the program we have written, we tried to develop a prototype of an analysis tool, that should be able to commit linguistic analysis based on a rule sheet automatically. We believe that there is a real need for such a program since it would allow both the student and the teacher to keep track of the student's changing language level in an easy and uncomplicated manner. The aim of the program is to provide one with an easily understandable single measure on a scale, but also showing the mistakes or certain grammatial constructs in the text.

2 Language learning and linguistic development

Both children learning their native language and adults learning a foreign language acquire the language step by step (Clahsen, 1986), (Schlyter, 2003), starting with nouns and simple phrases, advancing via using simple everyday langauge to (in some cases) using the complex language found in e.g. magazine and journal articles.

However, there is one substantial difference between native- and foreign-language acquisition: in native-language learning, there is no other "deeply-rooted" language with which the new language could interfere, whereas there is in foreign-language learning. This interference makes it difficult for Swedish-speaking FLE[1] learners to grasp e.g. word order, elision, complex relatives (e.g. *auxquels*) incflecting verbs after person and verb forms not present in the Swedish, e.g. *futur simple*, *conditionnel*[2], *gérondif* and *subjonctif*[3] These parts of the French language are (in general) only partially comprehended by casual (non-academic or non-professional) users. Of course, this applies to Swedish-speaking persons, who have not grown up in a French-speaking environment.

Below the levels are roughly described:

2.1 Novice (level 1–2, CEF (*C*ommon *E*uropean *F*ramework) A1–A2)

Novice users having only "survival-level" language skills tend to use mostly noun phrases in their communication, and do not yet inflect verbs fluently. Also, negations are usually of the type `<negator> <noun phrase>`. Novice-level students also need a co-operative and patient interlocutor who does not mind the student having limited fluency and vocabulary as well as needing to ask aid questions. An example of novice-level language production is *non, pas en train, en autobus*.

[1]French as a foreign laguage

[2]In Swedish, the conditional is (usually) constructed with the modal verb *skulle* + infinitive; consider it analogous with the English would+inf.

[3]That mode is certainly not used in the same way in Swedish (or German) as in French.

2.2 Intermediate-level user (level 3–4, CEF B1–B2

Intermediate-level users possess a basic command of the standard langauge, and have adequate ability in coping with day-to-day communication. About 50 % of their utterances consist of full sentences, and verbs are inflected and negated, but not always correctly. Swedes at this level often forget that e.g. *au* and *des* are maps of *à le* and *de les*, respectively, and therefore frequently use the tautological construct *au l'* or forget mapping *de les* to *des*. An intermediate-level user would say something like *Non, je ne suis pas venu en train, mais en autobus.*

2.3 Advanced user (level 5–6, CEF C1–C2)

After several years of French studies or stay in a French-speaking environment one can have reached the advanced level. This level is characterized by full acquisition of typical French-language constructs and productive use of a significant amount idiomatic expressions and fixed phrases (approaching native-speaker quality). Grammatical (syntactic) mistakes are rare and even complex language, like the one being used in this report, is (at least partially) mastered. Level 5+ example: *Si je dis quelque chose auquelle je ne suis pas tout à fait sûr, ce sera la composistion des rapports comme celui-ci. En réfléchissant, je pense que traduire ce rapport en français serait un bon exercise pour le lecteur qui se croit vraiment maîtriser le français. C'est improbable que personne qui lise ceci ne puisse le faire,* with any mistakes corrected.

3 The program

One of the major goals of our program was the separation of the above mentioned linguistic rules from the program logic itself. If we are able to separate language specific linguistic rules for analysis from the rest of the program, we are also able to reuse the same program for different languages. Of course, rule sheets must exist for every language to be checked against by our program. This part has to be done by linguists. The program itself should be able to load a new text, analyze it using the rules and show the results during runtime, and not during compile time as far as possible. We do not want a poor linguist having to learn the depths of programming languages before being able to translate his/her rules to a format, which the program is able to handle.

We therefore chose the growing standard XML to encode the rules. Further, we need a dictionary to look up words in the foreign language to obtain some grammatical information about every word. In order to analyze a text we had to split it up in words and sentences first. We decided to use Java's (version 1.4) ability to work with regular expressions. Although Perl and Prolog at the moment still tend to be the standard languages, their not providing one with integrated features for creating user interfaces like Java's Swing classes do is a severe disadvantage. It would certainly be possible to use e.g. Prolog for the parser, but since our program is likely to be used mostly on Windows machines target and non-programmers would prefer not having to install Prolog just for our program, we decided to do it entirely in Java.

The program on a high-level user perspective is relatively simple. Fist, the user chooses a file to load into an editor pane (or types or pastes[4] it directly into the editor), presses the "Analyze" button and then receives back another window, where grammatical mistakes are marked up and his/her language level is shown on a scale[5]

Looking deeper into the program, as soon as the analyze button is pressed by the user, the input text is sent to the core `Analyzer` as a string and the `Analyzer`begins its work. If it has not done it before, it loads the XML-encoded rules, splits up the text into tokens and applies the rules one after another to the tokens. Grammatical information about the words is found in a dictionary. In our case the dictionary is contained in a hash table, but better alternatives as using a database or a letter tree could be thought of.

When the analysis is done, the `Analyzer` returns the text to the GUI part of the program, but annotated with tags, for instance HTML[6]. The GUI then presents the result to the user again.

We will now take a deeper look into the program's different pieces.

4 Implementation concepts

4.1 Implementation language

Our program is implemented in 100 % Java and should therefore run on any platform supporting Java 1.4 and Swing GUIs.

4.2 The GUI

The GUI is a multilingual text editor with support for reading and writing Unicode text file as well as OS native clipboard handling, e.g. copy from MS Word and paste into the editor. After the user has typed, pasted or loaded a text, he/she can press the Analyze button to obtain a version of his/her text with XML tags, added by the `Analyzer` according to the rules specified in the rule file.

4.3 The rule file

The information of how a text has to be parsed by the `Analyzer` is stored in an XML-encoded file. We will call it the "rule file" or "XML rule file". (We assume that the reader has a basic knowledge of the current XML standards, understands roughly the syntax and semantics of XML and knows how XML and a DTD are connected one to the other.)

The strict separation between encoding the rules on the one hand and the `Analyzer`'s program logic on the other hand results in a gain of independency and configurability. The rules themselves may be changed easily without any need of complementary changes to the program logic. Since all the tags, their possible attributes and cardinalities are defined in a DTD file, the user has clear guidelines to follow when writing new rules. An invalid XML rule file will result in the parser throwing an error and writing an error message to the screen because a flag is set to check the XML's validity before parsing.

[4]Pasting is OS native and allows the user to analyze e.g. Word documents by opening them in Word, copying the text, and pasting it into the editor.

[5]We have not yet agreed how to report a preliminary language level using the data from the counters, so the table in (Schlyter, 2003).

[6]In our prototype only the raw XML is shown, but adding a style sheet would enhance the user's experience considerably. However, we consider it out of the scope of an 80-hour project, but may add it in a future version, if it is requested by the users and either of us finds the time.

As its name indicates, a rule file contains a set of different rules, each one encoded with valid XML tags. Each time the `Analyzer` is started, the rule file is read in, parsed by a standard XML DOM parser, transformed to a set of Java objects, which are then stored in a hash table. The `Analyzer` finally works with this hash table.

In the rule file, only valid XML tags, which are defined in the corresponding DTD, are allowed. A rule file may contain comment tags (starting with `<!--` and ending with `-->`, however such comments are simply ignored by the parser.

The rule file starts with the standard XML header. We used W3C-compliant version 1.0 standard XML. Currently, no namespaces are in use, however Xerces-J, our parser, should be able to handle namespaces correctly, but such functionality has not been tested out.

An important issue not to be forgotten in the header tag is choosing an appropriate character set. As we are dealing with French, the character set must at least contain all French special characters, such as *é* or *à* etc., and the rule file itself must be stored in the chosen format. Of course one has also to make sure, especially for unusual character encodings, that the used parser engine is able to handle the chosen character set. If an inappropriate character encoding set is chosen, there is a risk that certain words will never be found in the dictionary or the program will not be able to successfully match certain words in the text to the search rules (see below).

Further, the rule file consists of a set of "counter tags", two special rules and finally a set of the common or core rules.

4.3.1 The counters

The tag `counters` embraces a set of empty tags `counter`. These counters can be specified to be the holders of search information, in other words how many times certain grammatical forms are met by the `Analyzer`.

4.3.2 The Sentence Tokenize Rule

The first special rule is called the `sentenceTokenizeRule`. It looks like:

```
<sentenceTokenizeRule>
<regex>[.;:!?]</regex>
</sentenceTokenizeRule>
```

Each Sentence Tokenize Rule contains exactly one "regex-tag" which encloses a regular expression. The text will be split up into sentences using this regular expression as a delimiter. Therefore one has to be aware of blank characters, spaces and so on. (This is valid for all the tags containing regular expressions in the rule file.)

4.3.3 The Word Tokenize Rule

The second special rule is called the `wordTokenizeRule`. It is nearly the same as the `sentenceTokenizeRule` and looks like:

```
<wordTokenizeRule>
<regex>[a-zA-Z0-9]+</regex>
</wordTokenizeRule>
```

This rule is used for tokenizing a sentence into words.

4.3.4 The common rules

Following the two special rules come the common rules (or core rules). There must always exist at least one rule, which at the same time is the root or initial rule.

Each rule consists of two parts: The `search` and the `action`. A rule is applied by first searching in a sentence for an occurrence of the search criteria and then, dependent on whether a construct in the sentence (e.g. a word which) matches it or not, the `action`'s `found` respectively `notfound` part is executed.

The rule tag contains an attribute id, which is its unique identifier in the XML document ("is of type `ID`"). Additionally, each rule tag must specify a `framesize` value, indicating the number of words that the rule should be applied to. The start position for a rule's word frame to be opened is always the current position of the `Analyzer` (the left index) counting as many words to the right as the frame-size's value specifies or to the end of a sentence.

The Search A `search` can contain three different search criteria or any combination of the three.

- `<regex>apprends?</regex>` searches for regular expressions such as `apprend` or `apprends`

- `<lemma>apprendre</lemma>` searches for every word that has a lemma equal to the string *apprendre*, for instance *apprendrait* or *appris*

- `<inflection category="verb">...</inflection>` is able to search for certain grammatical constructs, here for instance for a verb.

The attribute `category` represents a word category and may take one of the following values: `noun`, `verb`, `adjective`, `pronoun`, `int_pronoun` (interrogative pronoun), `determiner`, `adverb`, `preposition`, `conjunction`, `numeral`[7], `interjection`, `abbreviation`, `residual` (any word that does not fit into one of the other categories).

Depending on the category, different combinations of tags may be added to the inflection tag. The program does not check whether a given combination of tags makes sense or not, it simply searches for this very combination (thus searching for a noun with a third person singular tag will of course never return any result). Basically, every word may have a combination of the following grammatical information:

1. `gender`: Possible values are `feminine` or `masculine`.

2. `number`: Possible values are `sg` (singular) or `pl` (plural).

3. `person`: Possible values are `1`, `2` or `3`.

4. `tense`: Possible values are `future`, `present`, `imperfect` or `past`.

5. `mode`: Possible values are `indicative`, `conditional`, `infinitive`, `participle` or `subjunctive`.

All these have a common empty tag format as `<tense value="present"/>`.

By writing these categories we oriented ourselves not only on purely grammatical and linguistic knowledge but tried to find a compromise with the needs of programming purposes too. Further,

[7] Numerals are rarely used in the dictionary. Our program will treat them as Residuals.

Category	gender	number	person	tense	mode
`noun`	X	X	-	-	-
`verb (ordinary)`	-	X	X	X	X
`verb (participle)`	X	X	X	X	X
`adjective`	X	X	-	-	-
`pronoun`	X	X	X	-	-
`int_pronoun`	X	X	X	-	-
`determiner`	X	X	-	-	-
`adverb`	-	-	-	-	-
`preposition`	-	-	-	-	-
`conjunction`	-	-	-	-	-
`numeral`	-	-	-	-	-
`interjection`	-	-	-	-	-
`abbreviation`	-	-	-	-	-
`residual`	-	-	-	-	-

Table 1: Possible tag combinations for given word categories

be aware of the order! The XML parser does only accept tags appearing in this specified order, thus for any rule you do not specify a gender, tense and mode, you have to assure that the number tag appears above the person tag.

It is also easy to recognize that in the rule file combinations of these tags can be specified that have no connection to real grammatical word forms at all. The program will not show any error message to the user but simply not return any result. Table 4.3.4 indicates the use of word categories and the corresponding grammatical information that may be specified.

Some comments have to be made:

1. In verbs, the grammatical information to be specified varies heavily. Whereas “normal” verb forms do not have any gender, participles may indeed have a gender. Imagine the feminine plural past participle of the French verb *ouvrir* (= to open) *ouvertes*. To have appropriate knowledge about this word, one needs to be able to specify a gender (feminine), a number (plural), a tense (past) and a mode (participle). On the other hand, of course a word like third person singular present of *rire* (= to laugh) does not have any

gender at all.

2. For many European languages, there are no participles with forms other than present or past.

3. Some pronouns do have a gender such as *he, she, his, her* while others, such as *I, you, we, my, yours,* do not. The same applies to interrogative pronouns.

4. Determiners that have a gender and/or a number are for example the French words *sa, son, ses, seize* (sixteen) and others.

Every word has at least a regular expression that may be searched for, namely the word itself, and a lemma. If one would like to look for a verb without further grammatical information, then at least a category can be specified.

Word categories are a subject of debate through the field of linguistics; often they are simply more or less given by the dictionary one uses.

An example:

```
<rule id="myRule" framesize="max"/>
<search>
<!-- <regex>(ouverte)|(ouvertes)
</regex> -->commented out!!
<lemma>ouvrir</lemma>
<inflection category="verb">
<gender value="feminine"/>
<number value="pl"/>
<tense value="past"/>
<mode value="participle"/>
</inflection>
</search>
```

These criteria will look in a frame of 4 words for the first occurrence of a word that has *ouvrir* as its lemma, is a verb and a feminine plural past participle. In the sentence *Il a fermé toutes les portes ouvertes* the word *ouvertes* will be found, but if the frame size is reduced to only 4, even though starting at the beginning of the sentence, applying the rule once to the sentence will result in nothing being found.

The action part specifies mainly three things:

1. Whether any counter should be incremented,

2. Whether tagging of current frame should be done and which tag should be used,

3. Which rule should be the next one to take

The rule uses `nextrule`'s value, which must point to an existing rule's `id`, to detect which rule to take next. This next rule does not need to be specified as child tag inside `found/notfound`. The only reason for which rules can be specified inside the `found/notfound` tags is the higher readability of the XML file. The program will always identify the next rule to take using the reference that `nextrule` points to, but disregard where in the XML file it is specified. If no next rule is set, then the initial or root rule will be the next one to be applied again.

If tagging is set to `yes` but no tag name is provided, the program will tag the output text with the current rule's `id`.

`action` always specifies what has to be done, if the `search` delivers a result, this is called the found and what has to be done if it does not find any result, simply called the `notfound`.

Such a structure makes it possible to search for multi word constructs or to compare words. For instance, one could specify a `search` that is looking for first person singular *je*, then specify an `action found` to go to another rule that then looks for a verb in a specified frame that also has a first person singular ending. On the other hand, if *je* is not found, then simply the word should be skipped and the second rule should never be called. This can be specified by not providing `notfound` with and next rule.

An example:

```
<action>
<found inccounter="fem_pl_pp_ouvrir"
nextrule="rule2"
dotagging="yes"
tagname="fem_pl_pp">
<rule id="rule2">...</rule>the next rule
</found>
<notfound dotagging="no">
</notfound>
</action>
```

These instructions will, if something is found by the `search`, cause the `Analyzer` to increment the counter `fem_pl_pp_ouvrir`, tag the frame with the tag `<fem_pl_pp>...</fem_pl_pp>` and then go to `rule2`. If nothing is found, the `Analyzer` simply goes to the next rule, which is not specified here, thus making the initial rule be applied again.

Overview
On the next page is an overview of all the possible tags, their attributes and possible child tags.

Tagname	**Attributes**	**(Direct) child tags**
Action		found notfound
Counters		counter*
Counter	id::ID # Required	
Found	inccounter::IDREF # IMPLIED nextrule::IDREF # IMPLIED dotagging::(yes | no) "no" tagname::NMTOKEN # IMPLIED	rule?
Gender	value::(feminine|masculine) # REQUIRED	
inflection	category::(noun|verb|adjective| pronoun|int_pronoun|determiner| adverb|preposition|conjunction| numeral|interjection| tense? abbreviation|residual) # REQUIRED	gender? number? person? mode?
Lemma::# PCDATA		
Mode	value::(indicative|conditional |infinitive|participle| subjunctive) #REQUIRED	
Notfound	inccounter::IDREF # IMPLIED nextrule::IDREF # IMPLIED dotagging::(yes|no) "no" tagname::NMTOKEN # IMPLIED	rule?
Number	value::(sg|pl) # REQUIRED	
Person	value::(1|2|3) # REQUIRED	
Regex::# PCDATA		
Rules		counters? sentenceTokenizeRule wordTokenizeRule rule+
Rule	id::ID # REQUIRED framesize::CDATA # REQUIRED (can take the value "max" for a frame as big a possible, or a numer > 0 alternatively).	search action
Search		regex? lemma? inflection?
SentenceTokenizeRule		regex
Tense	value::(future|present| imperfect|past) # REQUIRED	
WordTokenizeRule		regex

Table 2: All possible tags, their attributes and possible child tags

All tags that do not embrace child tags are by definition empty tags. Exceptions are the two tags `regex` and `lemma`, which simply only embrace `PCDATA`.

This table is to be read as: "There is a tag called `gender` that has exactly one attribute called `value`. A value of `value` is required and must be either `feminine` or `masculine`. The tag `gender` contains no further tags, therefore it is an empty tag."

`# REQUIRED` and `# IMPLIED` are used in their original XML-standard meaning. Child tags followed by a question mark `?` indicate that the current tag may have 0 or 1 of this child tag, a plus `+` that it may have 1 or more child tags of this kind, and an asterisk `*` that it may have 0 or more child tags. If no sign is specified, then exactly 1 child tag has to be added.

4.4 The Analyzer

The `Analyzer` carries out the core functionality of the program. Generally said, it holds a hash table of rules, as specified in the rule file, and simply applies one after the other sequentially to the text. It always starts by applying the initial rule and then following the rule's references (as specified by the `nextrule` attribute of the `found/notfound` tags). Also, as specified by the rule file, it increments counters if needed that may later on be read out and used for further purposes.

An `Analyzer` must implement the `IAnalyzer` interface, which only provides one method:
`public String analyze(String input)` `analyze()` takes the text to be parsed as an input `String` and returns a pseudo HTML-annotated `String` if some rules are specified to do some annotation tagging. This return `String` may in a further step be displayed by a web browser or processed further.

The class `Analyzer` implements this interface. Further, it provides a method
`public Map getCounters()` that returns a (Java) `Map` containing the counters. The key to the `Map` is simply the name of the counter (thus a `String`), while its value is of type `Integer`. The `Integer` object can simply be read out by using `intValue()` to receive an `int`.

The basic text analyzing algorithm works as follows:

1. Split up the input string into sentences by using `sentenceTokenizeRule`.

2. The basic algorithm to analyze a text is the following:

3. Split up every sentence into words by using `wordTokenizeRule`.

4. Repeat as long as there are more sentences:

   (a) Repeat as long as there are more words in a sentence:

      i. Go to the next word in the frame.

      ii. Try whether the current word matches the search criteria of the current rule.

      iii. If there is a match, then process the found part of the `action`.
      Otherwise, check whether there are still more words in this frame to be considered. If there are, choose the next one and go to 4a. If there are no more words in this frame, then the `search` was not successful. Process the not found part of the `action` in this case.

      iv. Eventually: Increment counters now and annotate the current frame with annotation tags if specified.

      v. Check for the next rule.
      If no such next rule is specified explicitly at this moment, then choose the initial rule again. The initial rule should be applied now to the first word in the sentence after the current word.
      Otherwise, apply the next rule to the current word.

   (b) Save the return value of applying the rule. This value now indicates the index position of the next word to be analyzed. This is now your current word. Go to 4a

5. Put all the possibly annotated/tagged words together to a single text again and return the final result string.

As we can see, this algorithm simply applies the initial rule sequentially to all words and if it once is successful, then it tries to process the next rule in the engine/rule tree to the word.

The `Analyzer` always works with word frames of a certain size that must be given in the rule tag. The frame's size should be set with care. If a frame is too small, certain grammatical constructs in a sentence, for instance words that are separated through other words but belong together, may not be found. Imagine the sentence *I have of course never seen something like this before.* If the frame size is set to a size of 3, the `Analyzer` is not able to connect logically the participle *seen* to the earlier encountered main verb *have*. On the other hand, if the frame size is too big, certain nonsensical forms may be believed to match the search criteria perfectly, although of course they do not.

4.5 The dictionary

We use a French dictionary[8] to look up inflections and lemmas of words. At program start, the whole dictionary is loaded into the computer's memory and stored in a simple hash table, which then is being used by the `search`. Each word in the text builds a key; the corresponding value is an object that is holder of all possible inflections of the word.

Due to the fact that a dictionary can never contain every possible word of a language and also depending on the tokenizing rules, it might be that certain words in the text are not found in the dictionary. The `Analyzer` will simply ignore this word and continue with the next word, thus never recognizing it as a match for the `search`. Such a behaviour should be tolerable as long as only a small percentage of the text's words are not contained in the dictionary.

5 Summary

Our program is surely not thought as a commercial product, but this has never been our goal. Indeed, with our prototype we have shown that basically it is possible to encode the linguists' rule sheets in a way, that they can more or less easily be changed independently of the rest of the source code without the need of recompilation. We used XML for this purpose.

The program needs a French dictionary to work with. At the moment, the availability of such dictionaries is not exactly the same as for English language. As our time resources were limited, we chose the easiest possible way to extract information from the dictionary. Of course, there is variety of better, but more complicated approaches from using databases to letter trees that improve memory usage[9] very significantly without great losses of performance.

A question still open is about the quality of analysis, which our program can reach. On the one hand, the more sophisticated the rules get, the more precise the results are. On the other hand, there will always be certain problems that cannot be answered fully either by the programmer or the linguist. What is the optimal word frame size for each rule, for example? An important point is that our program does not take into account any semantical information at all about the text to be analyzed, nor is a corpus illustrating *le bon usage* (of today) being used. However, we believe that extending our program to a semantical level too would be a much more demanding task, and it is dubious whether a semantical analysis would honour our huge efforts implementing it with any remarkably higher feedback quality.

Although our idea of splitting up the analyzation of repeatedly taken new `search`es and then `action`s, it is unceratin that such an approach really can cover the complexity of all linguistic rules. It was merely hard to find a least common general structures behind all rules. Of course, this structure now could be improved. One could think of adding several `search` parts to a single rule in the XML file, that then could be linked logically using an AND or an OR operator ("The whole `search` is only successful, if `search`1 AND `search`2 OR `search`3 are successful"). It would also be nice to have a graphical user interface for editing those rules.

[8]The dictionary is freely available for non-commercial purposes at `http://abu.cnam.fr`.

[9]At the moment the program uses up to 85 MB of RAM.

References

[Clahsen1986] Harald Clahsen. 1986. *Die Profilanalyse*, volume 1. Springer-Verlag, Berlin, Germany.

[Schlyter2003] Suzanne Schlyter. 2003. Stades de dveloppement en français l2. Avaliable at `http://www.rom.lu.se/durs/STADES_DE_DEVELOPPEMENT_EN_FRANCAIS_L2.pdf`.

Installation information

A Java prerequisites

Our program makes use of newer Java features to deal with regular expressions extensively. We ourselves used the J2SDK version 1.4.2, Standard Edition, to develop the program. Sun Microsystems introduced regular expression handling in version 1.4, so this is the oldest Java version that we can recommend to the user.

The latest Java 2 Software Development Kit and Java Runtime Environment can be downloaded from `http://java.sun.com/`.

B Insufficient initial heap size

Our program uses lots of memory! Be aware that if you start the program without defining special options for the Java Virtual Machine to increase the maximum heap size, the program might start to save information from the dictionary to the hard disk drive temporarily, not ending to write data to the disk. Under Windows implementations, you should therefore start the Java Virtual Machine as:
`java -mx90M ...`
This will set the JVM to use a maximum heap size of 90 megabytes which will be enough for our program to run. In our experiences, the program never used more then 86 MB or memory.

C Working with the Xerces-J XML DOM Parser

To parse the XML file, we used the freely available Xerces-J XML DOM parser. The program works fine with version 2.6.0. It can be found under: `http://xml.apache.org/xerces2-j/index.html` It is easiest to start your program, adding the corresponding paths by using the `-classpath` option. You need to add both the files `xercesImpl.jar` and `xml-apis.jar` to your `CLASSPATH`. Be aware that as soon as you set you these options, it might be that you have also to add the current working directiory (where your program source is located, for instance `se/lu/rom/sources`) and perhaps also the directory, where your binary runtime executables (`java.exe` under Windows), for instance `C:\Program files\jdk1.4.2`, are located.

```
java -classpath C:\...\Xerces-J\xercesImpl.jar;
                E:\...\Xerces-J\xml-apis.jar;
                [C:\...\jdk1.4.2\bin;
                C:\...\se\lu\rom\sources;]
                se.lu.rom.sources.MyMainClass
```

D Usage of the program classes

There is the central interface IAnalyzer and the implementing class Analyzer. Always use these two to get an instance of the Analyzer, as in:

```
/* Get the path of the rules file, e.g. from args[0] */
String rulesFilePath = args[0];

/* Get the dictionary to look up words as a Map, e.g. a Hashtable. Imagine the path
ImportDictController idc =
new ImportDictController(new File(args[1]));
Map dictionary = idc.importDictionary();

/* Instantiate your analyzer */
```

```
IAnalyzer myAnalyzer =
new Analyzer(rulesFilePath, dictionary);

/* Now do the analyzation and catch the result */
String outputText = myAnalyzer.analyze(inputText);
```

Now use the HTML annotated outputText to be displayed in your output window, for instance your browser.

A part-of-speech tagger for Swedish using the Brill transformation-based learning

François Marier
Lund Institute of Technology
fmarier@uwaterloo.ca

Bengt Sjödin
Lund University
bengtsjodin@hotmail.com

Abstract

This paper describes an implementation of a Brill Part-of-Speech tagger for the CarSim project. It introduces the concepts of part of speech tagging and Brill taggers and presents some results measured with the given implementation. Given the high running time of the Learning algorithm, very few results are available and these limitations, along with some partial solutions, are discussed.

1 Introduction

1.1 Goal

Our task was to implement a Brill part-of-speech (POS) tagger for Swedish using Java as the programming language. The focus was on the learner part of the algorithm as it is the most complex part of such a tagger.

1.2 The CarSim Project

The CarSim project[1] is an attempt to visualize written accident reports. The report is translated into a symbolic template, which is then used to generate a three-dimensional animation. Our tagger is supposed to be used in the translation part of the CarSim.

This is an important part because the texts are to be processed automatically, and as such the tagging of the words is vital for the information to be extracted correctly. Since all language are essentially an encoding of information, the program needs to decipher and reform it in meaningful ways so that it can be further processed into the final state.

For example the program must find the nouns and pronouns in the text and properly realize which of the words refer to separate entities and whether they are static or moving objects. This is done by splitting the text into sentences where the words then are tagged with their part-of-speech. The result is then used both for detecting road objects and clauses, which are used to fill event structures. These two things are what is needed to fill out the template.

[1]See http://www.lucas.lth.se/lt/carsim.shtml

1.3 Part-of-Speech Tagging

Part-of-speech (POS) tagging is also called grammatical tagging. It is one of the most common forms of corpus annotation. The labeling of the words in a sentence with their lexical or word classes is called tagging. The POS is divided between the open and the closed classes: The open class words are Nouns, Adjectives, Verbs and Adverbs whereas the closed are Determiners, Pronouns, Prepositions, Conjunctions, Auxiliaries and Modals.

Some morpho-syntactic information can also be provided, marking the word as a proper plural noun or a singular comparative adjective. This captures most of the information that a word contains.

One of the biggest problems with labeling a word correctly is disambiguation, meaning to find the intended form of the word, e.g. the word "can" could be a modal or a noun. An annotated text may be used to improve lexicons and otherwise help with the understanding and learning of languages.

1.4 Kinds of Taggers

The progress of automated part-of-speech taggers has gone from handwritten rules to Markov probability chains and on to machine-built rules. The early attempts required a large amount of repetitive work. Even the taggers that use Markov chains, though they achieve a high correctness probability, nest their rules inside a large set of such chains making them unreadable by humans.

The main advantages of the Brill tagger, with its machine-built rules, is that its rules can be easily transformed into a readable form and thus increase the human knowledge base. Also, it can be trained on top of a more complex pre-existing tagger to help improve its accuracy.

2 The Brill Method

A POS tagger that uses a transformation-based error-driven learner technique is called a Brill tagger.

Such a tagger must initially be trained to be able to tag a text correctly. A manually annotated corpus is used as a training reference. An initial state annotator first processes the same text without the tags. This annotator can be slightly complex, using statistics derived form the corpus, or very simple, merely tagging everything as nouns.

The machine-annotated text will then be compared with the reference noting the differences as errors. The algorithm will then try to find the most effective tag transformation rule, based on the current errors.

The rules are based on a number of templates that contain the structures and variables. For every error one instance of each type of rule is created. The most efficient rule is the one that corrects the most errors, and that one is found by letting all the various rules be applied to the text, remembering the best one.

The resulting text is, in each step of the learning algorithm, what is used as a base in the next thus always reducing the number of errors until a threshold is reached. The resulting set of rules is ordered by the amount of errors they correct.

Here is some pseudo-code describing the learning algorithm:

```
Annotate every word in the text
 with the most likely tag.

DO
   Create index of errors in
    annotated text.

   FOR every value in error index.
      Create all possible rules.

   Discard duplicate rules.
   Find rule with the largest
    error reduction.
   Store the rule.
   Apply the rule to text for
    the next iteration.

WHILE best rule fixes at least
 (some threshold value) errors.
```

2.1 Error Threshold

The Learner has to consider rules that fix more errors than a certain threshold. We have set this value to be 2 after performing a few tests on a very small training corpus. It appeared as if increasing this number would decrease the accuracy of the tagger since it discarded too many rules. However, bringing this number down to 1 also decreased the accuracy of the tagger.

We believe that this is due to the fact that the Learner would learn too many “bad” rules that might fix 1 error in the training corpus but would introduce many more in the test corpus. After all, a rule that fixes only 1 error overall on a large corpus can hardly be considered as representing a linguistic feature.

3 Implementation Overview

Here is a description of all the classes (see Figure 1) contained in our implementation.

Note that all of these classes are part of the se.lth.cs.BrillTagger Java package.

3.1 Main Programs

- **Tagger**: Class that does the actual tagging given an input file and a list of rules.
- **Learner**: Learns rules from the corpus.
- **FrequencyExtracter**: Extracts the tag frequency of each word from the corpus

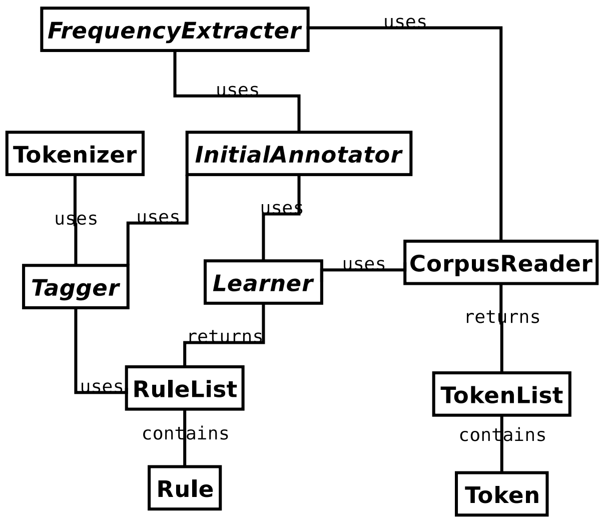


Figure 1: Class Diagram

- **InitialAnnotator**: The initial annotator takes a list of tokens and assign to them the most frequent tag.

3.2 Utility Classes

- **CorpusReader**: This class goes through the corpus, calling the ParseAction each time a word is encountered.

- **ParseAction**: Action to perform each time a new word-tag pair is extracted from the corpus by the CorpusReader. A typical action is to create a Token object containing the word and the tag.

- **RuleList**: Container for all instantiated rules learned by the Learner.

- **Token**: Basic unit of an annotated text.

- **Tokenizer**: Tokenizes a file into its grammatical components.

- **TokenList**: Basic data structure for manipulating an annotated text as a list of tokens (word and tag).

3.3 Rule Templates

- **Rule**: Base class for all the learned non-lexicalized rules.

- **RuleNotInstantiatable**: Exception thrown when there are not enough information to instantiate the rule. For example when trying to instantiate the PrevTagRule on the very first word of the corpus.

- **NextTagAndTwoBeforeRule**: Changes the current tag from source to destination if the next and second previous words are tagged in a certain way.

- **NextTagRule**: Changes the current tag from source to destination if the next word is tagged in a certain way.

- **NextTwoTagsRule**: Changes the current tag from source to destination if the next two words are tagged in a certain way.

- **OneOrTwoAfterRule**: Changes the current tag from source to destination if the next word or the one after that is tagged in a certain way.

- **OneOrTwoBeforeRule**: Changes the current tag from source to destination if the previous word or the one before that is tagged in a certain way.

- **OneOrTwoOrThreeAfterRule**: Changes the current tag from source to destination if the next word or the one after that or two words after is tagged in a certain way.

- **OneOrTwoOrThreeBeforeRule**: Changes the current tag from source to destination if the previous word or the one before that or two words before is tagged in a certain way.

- **PrevTagAndTwoAfterRule**: Changes the current tag from source to destination if the previous and second next words are tagged in a certain way.

- **PrevTagRule**: Changes the current tag from source to destination if the previous word is tagged in a certain way.

- **PrevTwoTagsRule**: Changes the current tag from source to destination if the previous two words are tagged in a certain way.

- **SurroundTagsRule**: Changes the current tag from source to destination if the previous and next words are tagged in a certain way.
- **TwoAfterRule**: Changes the current tag from source to destination if the second next word is tagged in a certain way.
- **TwoBeforeRule**: Changes the current tag from source to destination if the second previous word is tagged in a certain way.

4 User's Manual

4.1 FrequencyExtracter

The initial annotator must be run one time in order to generate the `tagfreq.dat` file that contains a hash of each word encountered in the corpus along with the statistically best tag. It is run in the following way:

```
  java se.lth.cs.BrillTagger.
FrequencyExtracter training
```

where "training" is the directory containing the files from the training corpus.

The program will display the number of words added to the hash table.

4.2 Learner

The Learner must also be run once before the Tagger can be applied to a text. It will output a list of rules into the `rules.dat` file. It is run in the following way:

```
  java se.lth.cs.BrillTagger.Learner
training
```

where "training" is the directory containing the files from the training corpus.

The program will initially display the corpus size and the baseline on the training corpus. Then it will display the selected rules along with the number of errors that they fix. Finally the program will print out the number of errors remaining as well as the number of rules that were learned and the accuracy on the training corpus.

4.3 Tagger

There are two ways to use the Tagger. One can run the tagger as a stand-alone program by passing the name of the file to tag:

```
  java se.lth.cs.BrillTagger.Tagger
test.txt
```

The program will then print each word along with its tag on the same line. There will only be one word (or token) per line.

The alternative is to call the Tagger from within a Java program. The Test.java class, distributed with the implementation, gives an example of such thing.

Basically, one has to use the `Tagger.tag(Reader reader)` method in order to have the Tagger read and tokenize the input text. The result is a TokenList which has a similar interface than ArrayList. Each token can then be extracted and used by a larger program.

5 Results

All of these results were gathered by running the actual Learner discarding rules having fixing less than 2 errors. The baseline on the test corpus was 82.542%.

In the first experiment, the full 13 non-lexicalized templates were used. The results follow:

nb words	nb files	nb rules	training accuracy	tagging accuracy
2336	1	25	96.147	83.032
4849	2	58	96.02	84.07
7136	3	82	96.118	84.212

Another experiment was performed where we only used the NextTag rule template. The results of this second experiment follow:

nb words	nb files	nb rules	training accuracy	tagging accuracy
4849	2	47	94.839	83.954
24345	10	123	95.046	84.544

6 Evaluation

We have implemented the Brill learning and tagging algorithms entirely. The initial annotator we are using is the one that derives the initial tags statistically. We have also implemented all 11 non-lexicalized rule templates mentioned in the original Brill paper, as well as two other ones mentioned in the Roche and Schabes paper ("previous bigram" and "next bigram").

Our implementation does not include the unknown word tagging rules nor the lexicalized rule templates.

6.1 Running Time of the Learner

The learning algorithm takes a very long time to run. Because of the way the algorithm works, there is no easy way out of this excessive run time. Here is a table of the training time for very small corpus sizes.

nb words	nb files	training time
2336	1	6 min
4849	2	55 min
7136	3	157 min

All tests were performed on a Pentium III 650 MHz with 256 MB of RAM.

7 Enhancements

We have considered two enhancements in order to cut down on the learning time. Without having access to a profiler, we decided to attempt to speed up the Learner by attacking one at a time the two suspected bottlenecks. More work on the learner will be necessary to identify other bottlenecks that might explain the performance of the implementation.

7.1 String Comparisons

The first attempt involved speeding up tag comparisons by converting string comparisons to pointer comparisons. We used the "intern()" method of the String class in order to achieve that. However, simple tests revealed that this modification had increased the learning time by approximately 40% on a very small training corpus.

After thinking about this surprising result, we came to the conclusion that string comparisons were probably quite fast in most cases since for the vast majority of tag comparisons, only 1 or 2 characters would have to be compared before the two tags would be deemed different. Hence the overhead involved in creating a hash table of String objects would increase the running time of the algorithm.

This modification is not part of the final implementation.

7.2 Corpus Cloning

In order to apply all rules to the same corpus, we decided to provide the TokenList class with a `clone()` method. That way, we do not have to undo a rule before applying the next one. Unfortunately, this also means that a large data structure is copied quite often.

So we decided to make the Token class immutable. This slows down some aspect of tagging/learning since modifying the tag of a Token now requires the program to create a brand new Token object. However, it also means that the cloning method of TokenList does not have to perform a deep copy anymore. It can just reuse the same Token instances instead of calling their copy constructor.

This modification has reduced the learning time by 20%. It is part of the final implementation.

8 Conclusions

Our initial tests revealed that the algorithm does bring an improvement over the baseline, but its running time is excessively large and it is hard to predict what accuracy we would get if the Learner was allowed to run on the entire corpus.

We were surprised by the slow learning rate of the algorithm since we were expecting a few rules to significantly improve the accuracy of the tagger on the test corpus.

More testing on a larger training corpus needs to happen before the tagger can be used effectively.

We can hope that the preliminary results that we have presented here will scale well with an enlarged corpus and will deliver the kind of accuracy that Brill claimed to get in his paper.

9 Acknowledgements

We are very grateful for Pierre Nugues' insightful comments and for the weekly meetings with him that have kept us on track and allowed us to progress even though we ran into some problems.

References

E. Brill 1995. *Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part-of-Speech Tagging.* Association for Computational Linguistics.

Emmanuel Roche and Yves Schabes 1995. *Deterministic Part-of-Speech Tagging with Finite-State Transducers.* Association for Computational Linguistics.

A probabilistic part-of-speech tagger for Swedish

Peter Nilsson
Department of Computer Science
University of Lund
Lund, Sweden
dat00pen@ludat.lth.se

Abstract

This paper presents a project for implementing and evaluating a probabilistic part-of-speech tagger for Swedish. The resulting program accurately tagged 95.9% of text containing only known words, and 89.7% of randomly selected text.

1 Introduction

Part-of-speech tagging is the method to assign each word in a text with a tag describing its proper part of speech in the context.

Two major problems in the process is resolving ambiguous words and classifying unknown words. To solve this task a few different methods has been developed. The most common are tagging based on rules, and probabilistic, or stochastic, tagging based on statistics. This paper presents a project for implementing a probabilistic part-of-speech tagger for Swedish.

2 Probabilistic POS tagging

In probabilistic part-of-speech tagging the tagger program is trained on a corpus annotated with a certain tagset. Statistics for the sequence of words and tags is collected, and is then used for estimating the most probable tagging of untagged text.

The background theory is given in (Charniak et al., 1993). An instructive description of the steps taken and decisions made in implementing an efficient stochastic part-of-speech tagger is found in (Carlberger and Kann, 1999), a paper which has served as a major guide for the initial steps of this project.

2.1 The basic model

This section contains a brief overview of the basic theory for developing the model.

A text can be considered as a sequence of n random variables $W_{1..n} = W_1, W_2, \ldots W_n$ where each variable can have a value from a fixed set of words $\{w_1, w_2, \ldots w_n\}$. For each sequence of word variables $W_{1..n}$ there is a corresponding sequence of random variables $T_{1..n}$, that can take their values from a fixed set of tags $\{t_1, t_2, \ldots t_n\}$. The tagging problem can then be described as finding the most probable sequence of tags, $t_{1,n}$, knowing the sequence of words, $w_{1,n}$. A formal definition of this is:

$$\mathcal{T}\left(w_{1,n}\right) = \arg\max_{t_{1,n}} P\left(t_{1,n} \mid w_{1,n}\right)$$

From Bayes' rule we know that:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}.$$

Applying this on the right hand side we get:

$$\mathcal{T}\left(w_{1,n}\right)=\underset{t_{1,n}}{\arg\max}\frac{P\left(w_{1,n}\mid t_{1,n}\right)P\left(t_{1,n}\right)}{P\left(w_{1,n}\right)}$$

and since the denominator will be constant for a given problem the expression is simplified to:

$$\mathcal{T}\left(w_{1,n}\right)=\underset{t_{1,n}}{\arg\max}\,P\left(w_{1,n}\mid t_{1,n}\right)P\left(t_{1,n}\right) \qquad (1)$$

2.2 Approximations for the implementation

The probability values needed for solving this problem is achieved from hand-annotated corpora. Even a large corpus will not be large enough for collecting the statistics needed in Equation 1. For this reason the problem is approximated using the following two assumptions:

$$P\left(t_i\mid t_{1,i-1},w_{1,i-1}\right)=P\left(t_i\mid t_{i-2},t_{i-1}\right)$$

$$P\left(w_i\mid t_{1,i},w_{1,i-1}\right)=P\left(w_i\mid t_i\right)$$

The first assumption states that the current tag is only dependent on the two previous tags. The second assumption states that the current word is only dependent on the current tag.

The expression for the problem now becomes:

$$\mathcal{T}\left(w_{1,n}\right)=\underset{t_{1,n}}{\arg\max}\prod_{i=1}^{n}P\left(t_i\mid t_{i-2},t_{i-1}\right)P\left(w_i\mid t_i\right) \qquad (2)$$

Equation 2 allows us to use statistics from every sequence of three tagged words, called trigrams, in a corpus. Even with this simplification the data will usually be too sparse to produce a reliable result. There are simply too many trigrams that will not appear or will only appear a small number of times in a corpus. We will have to approximate further by using the probabilities for the sequence of two tagged words, bigrams:

$$P\left(t_i\mid t_{i-2},t_{i-1}\right)\approx P\left(t_i\mid t_{i-1}\right)$$

and in the case the bigram data will be too sparse we will also use the unigrams:

$$P\left(t_i\mid t_{i-1}\right)\approx P\left(t_i\right)$$

Now we apply a useful strategy by interpolating between these:

$$P\left(t_i\mid t_{i-2},t_{i-1}\right)\approx$$

$$\lambda_1 P\left(t_i\mid t_{i-2},t_{i-1}\right)+\lambda_2 P\left(t_i\mid t_{i-1}\right)+\lambda_3 P\left(t_i\right)$$

where $\lambda_1+\lambda_2+\lambda_3=1$.

Putting this into Equation 2, we get:

$$\mathcal{T}\left(w_{1,n}\right)=$$

$$\prod_{i=1}^{n}\left[\lambda_1 P\left(t_i\mid t_{i-2},t_{i-1}\right)+\lambda_2 P\left(t_i\mid t_{i-1}\right)+\lambda_3 P\left(t_i\right)\right]P\left(w_i\mid t_i\right) \qquad (3)$$

Finding good values for the three lambdas is part of this project.

2.3 Estimating the probabilities

For estimating the probabilities in Equation 3 the following statistics is needed:

- Cn : the count of all words
- $C(w)$: the count of word w
- $C(t)$: the count of tag t
- $C(t_1,t_2)$: the count of bigrams where tag t_1 is followed by tag t_2
- $C(t_1,t_2,t_3)$: the count of trigrams where tag t_1 is followed by tag t_2 and then by tag t_3
- $C(w,t)$: the count of word w tagged with tag t

$$\mathcal{T}\left(w_{1,n}\right) = \underset{t_{1,n}}{\arg\max} \prod_{i=1}^{n} \left[\lambda_1 \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})} + \lambda_2 \frac{C(t_{i-1}, t_i)}{C(t_{i-1})} + \lambda_3 \frac{C(t_i)}{C_n} \right] \frac{C(w_i, t_i)}{C(t_i)}$$

Figure 1: Equation 4

The statistics is then used to estimate the probabilities as follows:

- $P(t_i) = \frac{C(t_i)}{C_n}$
- $P(t_i \mid t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}$
- $P(t_i \mid t_{i-2}, t_{i-1}) = \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})}$
- $P(w_i \mid t_i) = \frac{C(w_i, t_i)}{C(t_i)}$

Substituting this in Equation 3, we get the final equation in Figure 1.

This expression can easily be translated into a computer program. Tables for the various statistics can be built from an annotated corpus, and a suitable algorithm can iterate over the different combinations of probabilities to find the maximum. The naïve implementation, making an exhaustive search, will not suffice for sentences longer than a few words though, since the algorithms expansion is polynomial. The Viterbi algorithm is usually applied in this situation, as it is quite insensitive to any length of a sentence.

3 The SUC corpus

The corpus used is the SUC 1.0 corpus, created between 1989-1996 as part of a research project by the Departments of Linguistics at Stockholm University and Umeå University.

The selection of texts is a collection of modern Swedish prose first published in 1990 or later. Each of the more than one million words is annotated by the SUC tagset, which is listed in appendix A. The corpus is divided into 500 files containing roughly 2000 annotated words each.

4 The implementation

The program was developed in C++ using the Standard Template Library for the tables. It is divided into two major parts. The first part builds the statistics tables by parsing a set of annotated files. The second part implements the algorithm in Figure 1 and produces a suggestion of tagging for a text, one sentence at a time.

4.1 Training and evaluation

The goal for this project was to implement the basic model for probabilistic tagging in a computer program. The program was to be trained on the SUC corpus, and part of the corpus should be set aside for test and evaluation. Unknown words should be handled in a simple manner.

The corpus was divided into 50 randomly selected files each for the optimizing and test sets, and the remaining 400 for the train set.

A complete evaluation of the program was performed as follows:

The training phase consist of collecting the statistics from the train set. This means counting words, bigrams, trigrams etc and store the result in tables.

In the optimizing phase each text in the optimizing set is read and tagged by the program using the equation in Figure 1 with different sets of values for the three lambdas. The values used are between 0 and 1 in steps of 0.1. The tagging is then compared with the correct one, and of course the lambda values giving the best result are the optimal values for that particular training set. A sample output can be seen in appendix B.

During the test phase the test set is tagged in the same way, using the optimal lambda values, and the result is compared with the correct tags.

In the first evaluation session the program only handled words known from the test set. This was implemented in the optimizing and test phases by simply letting the program skip sentences that contained unknown words.

Two different sets of lambda values produced the same best result for the optimizing set. The values [0.5 0.4 0.1] and [0.7 0.2 0.1] for λ_1, λ_2 and λ_3 respectively each resulted in 95.90% correct tagging of the input. Investigating both these lambda settings for the test set resulted in 95.98% and 95.94% correct.

For the second evaluation session handling of unknown words was implemented as always tagging it as a noun singular. The tag selected from the SUC tagset was "NN UTR SIN IND NOM".

The optimal lambda values were [0.6 0.3 0.1] with a 89.85% correct tagging. The corresponding value for the test set was 89.24%.

To improve the result, the handling of unknown words was modified to use information from the bigram table built during the training. For each of the possible tags in the tagset the bigram table was searched for bigrams starting with that tag. From the collection of bigrams found, the one with the highest count was selected. In this bigram, the second tag was selected as the best candidate for tagging an unknown word following the first tag. This was put into a best-from-bigram table, shown in appendix C.

This time the optimal lambda values were [0.5 0.4 0.1] with 90.25% correct tagging. The corresponding value for the test set vas 89.73%. The output from the latter session is the content of appendix B.

4.2 Interactive session

This section will show a few examples of the program in interactive mode tagging text from the console.

```
> Jag ser en hök flyga.
Jag      PN UTR SIN DEF SUB
ser      VB PRS AKT
en       DT UTR SIN IND
hök      NN UTR SIN IND NOM
flyga    VB INF AKT
.         DL MAD
```

The input line is "Jag ser en hök flyga" ("I see a hawk fly") and the tagger outputs one line for each token with a suggested tag from the SUC tagset. All these words are known from the training, and the tagging is correct.

```
> Jag ser en falk flyga.
Jag      PN UTR SIN DEF SUB
ser      VB PRS AKT
en       DT UTR SIN IND
falk     NN UTR SIN IND NOM (?)
flyga    VB INF AKT
.        DL MAD
```

If we exchange "hök" ("hawk") for "falk" ("falcon") the tagger encounters an unknown word, as denoted by the question mark. In this case the best-from-bigram table is consulted, and a noun is suggested.

```
> Jag ser en glada flyga.
Jag      PN UTR SIN DEF SUB
ser      VB PRS AKT
en       DT UTR SIN IND
glada    JJ POS UTR/NEU PLU …[1]
flyga    VB INF AKT
.        DL MAD
```

Next, we change the bird to "glada" ("kite"). This word is also an adjective in Swedish ("happy/merry"). The training set contains the adjective form, but not the noun form. This results in the tagger finding the sequence Determiner/Singular – Adjective/Plural the most probable, even though it is a rare combination. In the test sentence the sequence is part or two trigrams, [VB DT JJ][2] and [DT JJ VB]. An

[1] This line is truncated. The tag is `JJ POS UTR/NEU PLU IND/DEF NOM.`

[2] These are abbreviations for brevity. The figures refer to the complete tags as shown in the example.

investigation of the tables shows that the count for those trigrams is 0 and 1. The bigram count is 1. If we investigate the corresponding values for the previous example, Determiner/Singular – Noun/Singular, the number of trigrams are 850 for [VB DT NN] and 49 for [DT NN VB]. The bigram count is 8189.

In fact, the word "glada" does appear once as a noun in the SUC corpus. If the tagger is trained on the full corpus it will then tag the test sentence above correctly. This of course a coincident, and the basic problem of incomplete information of the possible tagging for a word remains.

Using the console input it is interesting to test how well the tagger handles ambiguities.

```
> Ser du min min?
Ser     VB PRS AKT
du      PN UTR SIN DEF SUB
min     PS UTR SIN DEF
min     NN UTR SIN IND NOM
?       DL MAD
```

"Ser du min min?" ("Do you see the expression on my face?"). The ambiguous "min" is resolved in context. A sentence that is too difficult is not hard to construct:

```
> Var var var och en en gång?
Var     PN UTR SIN IND SUB/OBJ
var     VB PRT AKT
var     VB PRT AKT
och     KN
en      DT UTR SIN IND
en      DT UTR SIN IND
gång    NN UTR SIN IND NOM
?       DL MAD
```

"Var var var och en en gång?" ("Where were each and everyone once?"). Several words in this sentence have multiple tagging possibilities. Table 1 contains a list of the words, their possible tags and the number of occurrences of the combination word/tag. The tagger fails to catch the multiword expression "var och en" ("each and everyone"). After training the tagger on the full corpus the sentence was tested again:

```
> Var var var och en en gång?
Var     HA
var     VB PRT AKT
var     PN UTR SIN IND SUB/OBJ
och     KN
en      PN UTR SIN IND SUB/OBJ
en      DT UTR SIN IND
gång    NN UTR SIN IND NOM
?       DL MAD
```

This time the expression was noticed. Once again this is a coincidental, and it is easy to find similar cases where the tagger will fail. The difference between training from the train set and full corpus is near a 25% increase of parsed tokens (933629 vs. 1166896), and the resulting improvements are a simple verification of the fact that, in general, the larger the corpus the better.

Var		
	35	HA
	44	VB PRT AKT
	9	PN UTR SIN IND SUB/OBJ
	3	DT UTR SIN IND
	10	VB IMP AKT
var		
	5070	VB PRT AKT
	58	PN UTR SIN IND SUB/OBJ
	132	HA
	13	VB IMP AKT
	49	DT UTR SIN IND
	1	AB
	3	VB INF AKT
	3	NN NEU SIN IND NOM
och		
	25562	KN
en		
	13139	DT UTR SIN IND
	336	PN UTR SIN IND SUB/OBJ
	315	RG UTR SIN IND NOM
	3	UO
	8	AB
	2	NN UTR SIN IND NOM
gång		
	433	NN UTR SIN IND NOM

Table 1: Possible tags for words in the example sentence.

5 Conclusion

The goal for this project was to implement a model for probabilistic part-of-speech tagging in a computer program and evaluate it using the SUC corpus. The results of the training are very encouraging, considering that the tagging algorithm is more or less a straightforward implementation of the equation in Figure 1.

In the introduction the two main problems for part-of-speech tagging were mentioned, ambiguity and unknown words. Resolving ambiguity is part of the design of the probabilistic model, and as has been briefly demonstrated it generally work better the larger the training data is.

Handling of unknown words is a part that has to be added to the model. In the current implementation it is very rudimentary. Judging from the difference in result between the three evaluation sessions there is much to gain in improving this part of the program.

Another problem that appeared during testing is the problem of the program having an incomplete set of tagging options for a token. The tagger will produce a tagging based on available data, of course, but as shown in the example the result would sometimes have been better handled by the unknown word routine. This is a hard problem to solve, and would require some heuristics bridging tagging based on known words and tagging suggested by the handler of unknown words. A different, and complementary, approach is of course trying to avoid the problem by giving the tagger as complete training data as possible.

References

Carlberger, J. and V. Kann. Implementing an efficient part-of-speech tagger. *Software–Practice and Experience*, 29(9):815–832, 1999.

E. Charniak, C. Hendrickson, N. Jacobson, and M. Perkowitz. Equations for part-of-speech tagging. In *11th National Conf. Artificial Intelligence*, :784–789,1993.

A. The SUC tagset.

Category Code	Category
AB	Adverb
DL	Delimiter (Punctuation)
DT	Determiner
HA	Interrogative/Relative Adverb
HD	Interrogative/Relative Determiner
HP	Interrogative/Relative Pronoun
HS	Interrogative/Relative Possessive
IE	Infinitive Marker
IN	Interjection
JJ	Adjective
KN	Conjunction
NN	Noun
PC	Participle
PL	Particle
PM	Proper Noun
PN	Pronoun
PP	Preposition
PS	Possessive
RG	Cardinal Number
RO	Ordinal Number
SN	Subjunction
UO	Foreign Word
VB	Verb

Feature Code	Feature	
UTR	Common (Utrum)	Gender
NEU	Neutre	Gender
MAS	Masculine	Gender
UTR/NEU	Underspecified	Gender
-	Unspecified	Gender
SIN	Singular	Number
PLU	Plural	Number
SIN/PLU	Underspecified	Number
-	Unspecified	Number
IND	Indefinite	Definiteness
DEF	Definite	Definiteness
IND/DEF	Underspecified	Definiteness
-	Unspecified	Definiteness
NOM	Nominative	Case
GEN	Genitive	Case
SMS	Compound	Case
-	Unspecified	Case
POS	Positive	Degree
KOM	Comparative	Degree
SUV	Superlative	Degree
SUB	Subject	Pronoun Form
OBJ	Object	Pronoun Form

SUB/OBJ	Underspecified	Pronoun Form
PRS	Present	Verb Form
PRT	Preterite	Verb Form
INF	Infinitive	Verb Form
SUP	Supinum	Verb Form
IMP	Imperative	Verb Form
AKT	Active	Voice
SFO	S-form	Voice
KON	Subjunctive	Mood
PRF	Perfect	Perfect
AN	Abbreviation	Form

B. Output from test session,

unknown word = best-from-bigram

```
[ 0.0  0.0  1.0 ]    total: 116190, correct:98263, = 84.5710 %
[ 0.0  0.1  0.9 ]    total: 116190, correct:101528, = 87.3810 %
[ 0.0  0.2  0.8 ]    total: 116190, correct:102307, = 88.0515 %
[ 0.0  0.3  0.7 ]    total: 116190, correct:102820, = 88.4930 %
[ 0.0  0.4  0.6 ]    total: 116190, correct:103198, = 88.8183 %
[ 0.0  0.5  0.5 ]    total: 116190, correct:103441, = 89.0275 %
[ 0.0  0.6  0.4 ]    total: 116190, correct:103594, = 89.1591 %
[ 0.0  0.7  0.3 ]    total: 116190, correct:103741, = 89.2857 %
[ 0.0  0.8  0.2 ]    total: 116190, correct:103849, = 89.3786 %
[ 0.0  0.9  0.1 ]    total: 116190, correct:103920, = 89.4397 %
[ 0.0  1.0  0.0 ]    total: 116190, correct:103698, = 89.2486 %
[ 0.1  0.0  0.9 ]    total: 116190, correct:102115, = 87.8862 %
[ 0.1  0.1  0.8 ]    total: 116190, correct:102752, = 88.4345 %
[ 0.1  0.2  0.7 ]    total: 116190, correct:103157, = 88.7830 %
[ 0.1  0.3  0.6 ]    total: 116190, correct:103445, = 89.0309 %
[ 0.1  0.4  0.5 ]    total: 116190, correct:103641, = 89.1996 %
[ 0.1  0.5  0.4 ]    total: 116190, correct:103842, = 89.3726 %
[ 0.1  0.6  0.3 ]    total: 116190, correct:103937, = 89.4543 %
[ 0.1  0.7  0.2 ]    total: 116190, correct:104050, = 89.5516 %
[ 0.1  0.8  0.1 ]    total: 116190, correct:104106, = 89.5998 %
[ 0.1  0.9  0.0 ]    total: 116190, correct:104023, = 89.5284 %
[ 0.2  0.0  0.8 ]    total: 116190, correct:102868, = 88.5343 %
[ 0.2  0.1  0.7 ]    total: 116190, correct:103289, = 88.8966 %
[ 0.2  0.2  0.6 ]    total: 116190, correct:103576, = 89.1436 %
[ 0.2  0.3  0.5 ]    total: 116190, correct:103758, = 89.3003 %
[ 0.2  0.4  0.4 ]    total: 116190, correct:103927, = 89.4457 %
[ 0.2  0.5  0.3 ]    total: 116190, correct:104021, = 89.5266 %
[ 0.2  0.6  0.2 ]    total: 116190, correct:104103, = 89.5972 %
[ 0.2  0.7  0.1 ]    total: 116190, correct:104152, = 89.6394 %
[ 0.2  0.8  0.0 ]    total: 116190, correct:104023, = 89.5284 %
[ 0.3  0.0  0.7 ]    total: 116190, correct:103266, = 88.8768 %
[ 0.3  0.1  0.6 ]    total: 116190, correct:103612, = 89.1746 %
[ 0.3  0.2  0.5 ]    total: 116190, correct:103800, = 89.3364 %
[ 0.3  0.3  0.4 ]    total: 116190, correct:103941, = 89.4578 %
[ 0.3  0.4  0.3 ]    total: 116190, correct:104076, = 89.5740 %
[ 0.3  0.5  0.2 ]    total: 116190, correct:104166, = 89.6514 %
[ 0.3  0.6  0.1 ]    total: 116190, correct:104200, = 89.6807 %
[ 0.3  0.7  0.0 ]    total: 116190, correct:104046, = 89.5482 %
[ 0.4  0.0  0.6 ]    total: 116190, correct:103517, = 89.0929 %
[ 0.4  0.1  0.5 ]    total: 116190, correct:103787, = 89.3252 %
[ 0.4  0.2  0.4 ]    total: 116190, correct:103955, = 89.4698 %
[ 0.4  0.3  0.3 ]    total: 116190, correct:104070, = 89.5688 %
[ 0.4  0.4  0.2 ]    total: 116190, correct:104194, = 89.6755 %
[ 0.4  0.5  0.1 ]    total: 116190, correct:104244, = 89.7186 %
[ 0.4  0.6  0.0 ]    total: 116190, correct:104071, = 89.5697 %
[ 0.5  0.0  0.5 ]    total: 116190, correct:103704, = 89.2538 %
[ 0.5  0.1  0.4 ]    total: 116190, correct:103927, = 89.4457 %
[ 0.5  0.2  0.3 ]    total: 116190, correct:104051, = 89.5525 %
[ 0.5  0.3  0.2 ]    total: 116190, correct:104192, = 89.6738 %
[ 0.5  0.4  0.1 ]    total: 116190, correct:104258, = 89.7306 %
[ 0.5  0.5  0.0 ]    total: 116190, correct:104103, = 89.5972 %
[ 0.6  0.0  0.4 ]    total: 116190, correct:103817, = 89.3511 %
[ 0.6  0.1  0.3 ]    total: 116190, correct:104026, = 89.5309 %
[ 0.6  0.2  0.2 ]    total: 116190, correct:104157, = 89.6437 %
[ 0.6  0.3  0.1 ]    total: 116190, correct:104247, = 89.7211 %
[ 0.6  0.4  0.0 ]    total: 116190, correct:104089, = 89.5852 %
[ 0.7  0.0  0.3 ]    total: 116190, correct:103917, = 89.4371 %
[ 0.7  0.1  0.2 ]    total: 116190, correct:104118, = 89.6101 %
[ 0.7  0.2  0.1 ]    total: 116190, correct:104207, = 89.6867 %
[ 0.7  0.3  0.0 ]    total: 116190, correct:104086, = 89.5826 %
[ 0.8  0.0  0.2 ]    total: 116190, correct:103957, = 89.4716 %
[ 0.8  0.1  0.1 ]    total: 116190, correct:104148, = 89.6359 %
[ 0.8  0.2  0.0 ]    total: 116190, correct:104065, = 89.5645 %
[ 0.9  0.0  0.1 ]    total: 116190, correct:103966, = 89.4793 %
[ 0.9  0.1  0.0 ]    total: 116190, correct:104023, = 89.5284 %
[ 1.0  0.0  0.0 ]    total: 116190, correct:103242, = 88.8562 %
```

C. best-from-bigram table

Collected from the bigram table, built during training.The known tag points to the most common tag following it. Sorted by the number of occurrences in the train set.

```
 14442: NN UTR SIN IND NOM      -->>    PP
 12567: PP      -->>    NN UTR SIN DEF NOM
  8853: JJ POS UTR SIN IND NOM  -->>    NN UTR SIN IND NOM
  8830: IE      -->>    VB INF AKT
  8189: DT UTR SIN IND  -->>    NN UTR SIN IND NOM
  7524: NN UTR SIN DEF NOM      -->>    PP
  7235: VB PRS AKT      -->>    AB
  6547: <s>     -->>    PP
  6227: NN UTR PLU IND NOM      -->>    PP
  6160: JJ POS UTR/NEU PLU IND/DEF NOM  -->>    NN UTR PLU IND NOM
  5912: AB      -->>    PP
  5876: PM NOM  -->>    PM NOM
  5761: NN NEU SIN IND NOM      -->>    PP
  5052: PN UTR SIN DEF SUB      -->>    VB PRT AKT
  4524: DL MID  -->>    KN
  4377: PN NEU SIN DEF SUB/OBJ  -->>    VB PRS AKT
  3680: JJ POS UTR/NEU SIN DEF NOM      -->>    NN UTR SIN DEF NOM
  3678: HP - - -        -->>    VB PRS AKT
  3601: VB PRT AKT      -->>    AB
  3595: KN      -->>    NN UTR SIN IND NOM
  3546: DT NEU SIN IND  -->>    NN NEU SIN IND NOM
  3378: VB INF AKT      -->>    PP
  3329: DT UTR SIN DEF  -->>    JJ POS UTR/NEU SIN DEF NOM
  3213: JJ POS NEU SIN IND NOM  -->>    NN NEU SIN IND NOM
  3009: NN NEU SIN DEF NOM      -->>    PP
  2857: DL MAD  -->>    DL MID
  2846: PL      -->>    PP
  2563: RG NOM  -->>    NN UTR PLU IND NOM
  2510: NN NEU PLU IND NOM      -->>    PP
  2086: VB PRS SFO      -->>    PP
  2083: SN      -->>    PN UTR SIN DEF SUB
  2067: PS UTR SIN DEF  -->>    NN UTR SIN IND NOM
  2007: NN UTR PLU DEF NOM      -->>    PP
  1988: DT UTR/NEU PLU DEF      -->>    JJ POS UTR/NEU PLU IND/DEF NOM
  1717: DL PAD  -->>    DL MAD
  1683: DT NEU SIN DEF  -->>    JJ POS UTR/NEU SIN DEF NOM
  1513: UO      -->>    UO
  1490: AB POS  -->>    PP
  1456: VB INF SFO      -->>    PP
  1434: PN UTR PLU DEF SUB      -->>    VB PRS AKT
  1393: VB SUP AKT      -->>    PP
  1377: PN UTR SIN IND SUB      -->>    VB PRS AKT
  1339: PN UTR/NEU SIN/PLU DEF OBJ      -->>    PP
  1250: PC PRF UTR SIN IND NOM  -->>    NN UTR SIN IND NOM
  1232: NN UTR SIN DEF GEN      -->>    NN UTR SIN IND NOM
  1176: HA      -->>    PN UTR SIN DEF SUB
  1138: PM GEN  -->>    NN UTR SIN IND NOM
   970: PN UTR/NEU PLU DEF SUB  -->>    VB PRS AKT
   924: PC PRS UTR/NEU SIN/PLU IND/DEF NOM      -->>    NN UTR SIN IND NOM
   923: PS UTR/NEU SIN/PLU DEF  -->>    NN UTR SIN IND NOM
   893: VB PRT SFO      -->>    PP
   871: PS NEU SIN DEF  -->>    NN NEU SIN IND NOM
   849: PC PRF UTR/NEU PLU IND/DEF NOM  -->>    NN UTR PLU IND NOM
   821: VB SUP SFO      -->>    PP
```

```
725: JJ KOM UTR/NEU SIN/PLU IND/DEF NOM      -->>    NN UTR SIN IND NOM
721: NN NEU PLU DEF NOM      -->>    PP
681: IN      -->>    DL MID
673: PS UTR/NEU PLU DEF      -->>    NN UTR PLU IND NOM
619: PN UTR SIN DEF OBJ      -->>    DL MAD
607: PN UTR SIN DEF SUB/OBJ  -->>    VB PRS AKT
574: JJ POS UTR/NEU SIN/PLU IND/DEF NOM      -->>    NN UTR SIN IND NOM
557: PN NEU SIN IND SUB/OBJ  -->>    PP
517: NN NEU SIN DEF GEN      -->>    NN UTR SIN IND NOM
502: HP NEU SIN IND  -->>    VB PRS AKT
481: JJ POS UTR/NEU PLU IND NOM      -->>    NN UTR PLU IND NOM
444: PC PRF NEU SIN IND NOM  -->>    NN NEU SIN IND NOM
444: DT UTR/NEU PLU IND      -->>    NN UTR PLU IND NOM
436: NN AN   -->>    RG NOM
433: PC PRF UTR/NEU SIN DEF NOM      -->>    NN UTR SIN DEF NOM
425: PN UTR SIN IND SUB/OBJ  -->>    PP
417: AB KOM  -->>    KN
385: RO NOM  -->>    NN UTR SIN DEF NOM
359: NN UTR - - SMS  -->>    KN
343: NN UTR PLU DEF GEN      -->>    NN UTR SIN IND NOM
335: DT UTR/NEU SIN IND      -->>    NN UTR SIN IND NOM
312: DT UTR/NEU PLU IND/DEF  -->>    NN UTR PLU IND NOM
305: DT UTR/NEU SIN/PLU IND  -->>    NN UTR SIN IND NOM
301: PN UTR/NEU PLU DEF OBJ  -->>    DL MAD
290: JJ SUV UTR/NEU SIN/PLU DEF NOM  -->>    NN UTR SIN DEF NOM
289: PN UTR/NEU PLU IND SUB/OBJ      -->>    PP
285: JJ POS MAS SIN DEF NOM  -->>    NN UTR SIN DEF NOM
281: JJ POS UTR SIN IND/DEF NOM      -->>    NN UTR SIN IND NOM
243: AB SUV  -->>    PP
200: NN UTR SIN IND GEN      -->>    NN UTR SIN IND NOM
196: AB AN   -->>    RG NOM
175: VB IMP AKT      -->>    AB
156: RG UTR SIN IND NOM      -->>    NN UTR SIN IND NOM
146: PN UTR PLU DEF OBJ      -->>    PP
141: NN NEU - - SMS  -->>    KN
139: PN UTR/NEU PLU DEF SUB/OBJ      -->>    VB PRS AKT
131: NN UTR PLU IND GEN      -->>    NN UTR SIN IND NOM
130: NN NEU SIN IND GEN      -->>    NN UTR SIN IND NOM
126: HD UTR SIN IND  -->>    NN UTR SIN IND NOM
115: JJ POS NEU SIN IND/DEF NOM      -->>    NN NEU SIN IND NOM
108: NN NEU PLU DEF GEN      -->>    NN UTR SIN IND NOM
104: DT UTR SIN IND/DEF      -->>    NN UTR SIN IND NOM
103: RG UTR/NEU SIN DEF NOM  -->>    NN UTR SIN DEF NOM
 95: RG NEU SIN IND NOM      -->>    NN NEU SIN IND NOM
 92: JJ SUV UTR/NEU SIN/PLU IND NOM  -->>    PP
 92: HP UTR SIN IND  -->>    VB PRS AKT
 86: HD UTR/NEU PLU IND      -->>    NN UTR PLU IND NOM
 75: NN NEU PLU IND GEN      -->>    NN UTR SIN IND NOM
 59: KN AN   -->>    PM NOM
 58: RG SMS  -->>    KN
 56: JJ SUV UTR/NEU PLU DEF NOM      -->>    PP
 56: HD NEU SIN IND  -->>    NN NEU SIN IND NOM
 52: HS DEF  -->>    NN UTR SIN IND NOM
 33: NN - - - -      -->>    DL MAD
 33: JJ POS MAS SIN DEF GEN  -->>    NN UTR SIN IND NOM
 32: HP UTR/NEU PLU IND      -->>    VB PRS AKT
 31: PM SMS  -->>    KN
 30: NN UTR - - -    -->>    PP
 30: DT NEU SIN IND/DEF      -->>    NN NEU SIN IND NOM
 27: PC PRF MAS SIN DEF NOM  -->>    NN UTR SIN DEF NOM
```

```
27: NN - - - SMS    -->>    KN
25: JJ POS UTR - - SMS      -->>    KN
23: VB KON PRS AKT  -->>    PN UTR/NEU SIN/PLU DEF OBJ
22: PN MAS SIN DEF SUB/OBJ  -->>    VB PRT AKT
19: VB AN   -->>    PM NOM
14: VB KON PRT AKT  -->>    JJ POS NEU SIN IND NOM
14: JJ SUV MAS SIN DEF NOM  -->>    NN UTR SIN IND NOM
14: JJ POS UTR/NEU PLU IND/DEF GEN  -->>    NN UTR SIN IND NOM
14: AB SMS  -->>    KN
13: DT MAS SIN DEF  -->>    NN UTR SIN IND NOM
12: PL SMS  -->>    KN
11: JJ POS UTR/NEU SIN DEF GEN      -->>    NN UTR SIN IND NOM
 9: JJ AN   -->>    NN UTR SIN IND NOM
 8: RO MAS SIN IND/DEF NOM  -->>    NN UTR SIN IND NOM
 8: RG MAS SIN DEF NOM      -->>    HP - - -
 6: JJ SUV UTR/NEU PLU IND NOM      -->>    NN UTR PLU IND NOM
 5: JJ POS UTR/NEU SIN/PLU IND NOM  -->>    NN UTR PLU IND NOM
 4: RO GEN  -->>    NN UTR SIN IND NOM
 4: PC PRF UTR/NEU PLU IND/DEF GEN  -->>    NN UTR SIN IND NOM
 4: DT UTR/NEU SIN DEF      -->>    NN UTR SIN DEF NOM
 4: DT MAS SIN IND  -->>    NN UTR SIN IND NOM
 3: VB SMS  -->>    KN
 3: RG GEN  -->>    NN UTR SIN IND NOM
 3: NN NEU - - -    -->>    DL MAD
 3: JJ POS UTR/NEU - - SMS  -->>    KN
 2: RO SMS  -->>    KN
 2: PP AN   -->>    NN UTR SIN IND NOM
 2: PC PRF UTR/NEU SIN DEF GEN      -->>    NN UTR SIN IND NOM
 2: PC PRF MAS SIN DEF GEN  -->>    NN UTR SIN IND NOM
 2: JJ KOM UTR/NEU SIN/PLU IND/DEF SMS      -->>    KN
 2: JJ KOM UTR/NEU SIN/PLU IND/DEF GEN      -->>    NN UTR SIN IND NOM
 1: VB KON PRT SFO  -->>    DT NEU SIN IND
 1: VB IMP SFO      -->>    PP
 1: PC PRS UTR/NEU SIN/PLU IND/DEF GEN      -->>    NN UTR SIN DEF NOM
 1: PC PRF UTR SIN IND GEN  -->>    NN UTR PLU IND NOM
 1: PC AN   -->>    NN UTR SIN IND NOM
 1: JJ SUV MAS SIN DEF GEN  -->>    JJ POS UTR/NEU SIN DEF NOM
 1: JJ POS UTR SIN IND GEN  -->>    NN UTR SIN IND NOM
 1: HP NEU SIN IND SMS      -->>    KN
 1: DT AN   -->>    RG NOM
```

Morphar: A Morphological Parser for Swedish

Thomas Raneland
Lund Institute of Technology,
University of Lund
d00tr@efd.lth.se

Abstract

This paper introduces the model and implementation for Morphar, a morphological parser for Swedish. The parser approach is intended to be as simple and natural as possible, taking advantage of the characteristics of Swedish morphology. It is based around a lexicon and a parser inspired by compiler construction techniques. The reference implementation has shown the model to work. The program is fast and returns correct results for more than 70 % of random input words. The implementation is distributed freely for use in non-commercial applications.

1 Introduction

The purpose of a morphological parser is, given an inflected word, to analyse the word and provide the user with information on what the root word is, and what inflections it has undergone. Such a computer program may be a stand-alone tool, but is often used in conjunction with other language processing components to analyse complete texts. The parser discussed in this paper, Morphar, is intended for use in both kinds of situations.

Most morphological parsers today take the approach of the two-level model presented by Kimmo Koskeniemmi (Koskenniemi, 1997). The Morphar project takes a different approach. The intention was to use as natural a model as possible. Data structures are laid out much as in an ordinary non-electronic dictionary, with entries for each word, holding information on syntactic category and possible inflections as well as suffixes for compound forms. The parser is constructed as typical computer language compiler.

The report will discuss general morphology (chapter 2) and Swedish morphology (chapter 3). Then the theory will be used to model the morphological parser Morphar (chapter 4). Some important implementation notes are included (chapter 5). Finally, pros and cons of the model and implementation are discussed (chapter 6).

2 Morphology

Morphology is the study of *morphemes*, the minimal units of meaning in a language. There are two kinds of morphemes, grammatical morphemes and lexical morphemes. Lexical morphemes correspond to the word stems, while grammatical morphemes can be either grammatical words or affixes.

Furthermore, affixes can be divided into four groups: *prefixes* (before the stem), *suffixes* (after the stem), *infixes* (in-between parts of the stem), and *circumfixes* (surrounding the stem). Examples of prefixes are pre-, sur- and in- used as above, and examples of suffixes are -s and -ed.

In European languages, words are built up by one or more morphemes (Nugues, 2003). Often, a lexical morpheme that defines the meaning of the word is concatenated with a number of grammatical morphemes (prefixes and/or suffixes) that defines the semantic function in the phrase or meaning.

2.1 Inflection

Grammatical affixes are often added to a stem in order for the word to agree in tense, number, gender or case to its neighbour words in a meaning. This is called *inflection*.

Inflection is, in most languages, relatively predictable (Nugues, 2003). For instance, in English, plural is indicated by an -s suffix, and past tense for verbs is indicated by an -ed suffix. However, most languages include a number of exceptions from these simple rules: The plural of *sheep* is *sheep*, and the past tense form of *eat* is *eaten*.

2.2 Derivation

Another class of affixes are the *derivational* affixes. Such affixes, when added to a stem, may change the syntactic category and/or the meaning of the word. Examples of English derivational morphemes are prefixes un-, con- and suffixes -ly, -ist and -ish (Fromkin, 1998). Derivation rules can be combined (as in un-system-atic-al-ly, where un-, -atic, -al, and -ly all are derivational morphemes).

Unlike inflectional morphemes, derivation rules often have many exceptions. Furthermore, derivation is irregular; although the adjective *doable* can be derived from the verb *do,* no adjective **pleasable* can be derived from the verb *please*. There is no logical explanation for this, and hence no rule to decide when the rule may be applied.

2.3 Compounds

Combining words together may form new words. Such words are called *compounds*. The category of a compound word is the category of the last word. The last word is the only word that is inflected. However, the words may be "glued" together by *compositional morphemes,* such as in the Swedish compound *tidsmaskin* (time machine), composed of *tid* (time) *-s-* (compositional morpheme) and *maskin* (machine).

2.4 Paradigms

Since the inflectional system is rather predictable, one may construct patterns of inflections that apply to a class of words. For instance, In Swedish, many nouns with ∅-plural[1] use the -et suffix to denote definite form, and the -en suffix to denote both plural and definite form, e.g. *bord* (table), *bordet* (the table), *bord* (tables), *borden* (the tables). We may say that all such words share the same *paradigm*. The paradigm is an inherent property of the word.

[1] ∅ is the symbol for the "zero" morpheme. The ∅ morpheme does not change the textual representation of the word.

3 Swedish Morphology

The general morphology described in the previous chapter can be used directly when constructing the parser. However, most languages do not use all the possible features, and so the model can be simplified. As a first – important – example, inflections are only realized by suffixes in Swedish.

The Swedish language is built up by words from the following grammatical categories:

- Nouns,
- Adjectives,
- Pronouns,
- Numerals,
- Verbs,
- Adverbs,
- Prepositions,
- Conjunctions,
- Subjunctions,
- Interjections.

These categories should be well known, and so for the rest of this chapter, I will concentrate on special cases for Swedish and inflections.

3.1 Nouns

Swedish nouns may be inflected to agree in number, definiteness, and case. Number can be singular and plural, definiteness is definite or indefinite and case is either normal form or genitive.

An inherent property of Swedish nouns is gender, which may be neuter or the "common" gender (Dalgish, 2003).

The paradigms for nouns are called *declensions*. Swedish nouns can be categorized into four declensions: -or, -ar, -er, and ∅ declension (Hellberg, 1978). Each declension has several exceptions. Additionally, words may not belong to any of these declensions. Most such words are borrowed from other languages, e.g. English (*musical, cocktail*) and Latin (*examen, spektrum*).

3.2 Adjectives

Adjectives may have comparative forms: positive (the normal form), comparative, and superlative form. Depending on the function of the adjective, it may be inflected to agree with the noun (or pronoun) in number, gender and definiteness. The rules are rather complex, and there is no need to go into detail on these issues, so instead a list of possible inflections is presented. These are all the forms that an adjective can take:

- Common gender form,
- Neuter form,
- Plural form,
- Definite form,
- Masculine definite form,
- Comparative form,
- Superlative definite form,
- Superlative indefinite form.

The first five inflections are in the positive form. The comparative form cannot be further inflected. The superlative form may be definite or indefinite.

As you can see, definite singular positive form may be in the normal form or in *masculine* form. If the sex of the noun is masculine, the sex of the adjective may (but need not) be masculine. Examples: *den vackre/vackra mannen* (the beautiful man), *den vackra kvinnan* (the beautiful woman), *den vackra stolen* (the beautiful chair). As you might have noticed, sexless nouns always use the non-masculine form.

Many adjectives may be compared by adding -*are* and -*ast* to the normal form to get the comparative and superlative forms. These are the regular adjectives. Other adjectives are irregular, e.g. *liten–mindre–minst* (small–smaller–smallest) and *gammal–äldre–äldst* (old–older–oldest). Finally, many adjectives are compared periphrastically, e.g. *handikappad–mer handikappad–mest handikappad* (handicapped–more handicapped–most handicapped) or not at all, e.g. *död* (dead), *blind* (blind), and *tom* (empty). (Above examples taken from Stroh-Wollin, 1998.)

3.3 Verbs

The dictionary form of the verb is called the infinitive form. This form is often preceded by the infinitive marker (*att skriva*, to write). Verbs are inflected by tense (present, past, and supine) and mood (indicative, imperative, conjunctive), where the indicative form is the normal form. Conjunctive forms may be in the present or past form. The present form is no longer used. Therefore, the past conjunctive may be denoted just “conjunctive”.

The indicative forms may be divided into active and inactive. Active form is the normal form. Passive form is constructed by adding an -s to the stem, e.g. *skrämma–skrämmas, skräm–skräms, skrämde–skrämdes*.

The paradigm for verbs are called *conjugations*. There are four conjugations in Swedish. Conjugation 1–3 are weak and are easy to implement. The 4th conjugation is strong and may include ablauts.

Participle

Participle is sometimes considered to be a grammatical category of its own. In this theory, participles are considered to be inflected verbs. The present participle takes one form in Swedish. The past participle is inflected on gender and number. The three forms are common gender, neuter, and plural.

3.4 Other categories

The other grammatical categories are considered indeclinable in this theory. Despite this, some pronouns (e.g. possessive pronouns) are inflected to agree in gender and number just like adjectives, e.g. *min–mitt–mina*, (my/mine). Also, numerals can be divided into cardinals and ordinals. Despite this, each word in the closed categories is considered to be a lexeme in its own right.

4 The Morphar Model

The Morphar morphological parser is a lexicon-based compiler-inspired system. Every non-compound word is described in the lexicon, including all inflectional endings. When the system analyses a word, the word is looked up in the lexicon. During look-up, each part of the word is replaced by its description. At the end, we have a structured morphological description of the complete word. There are some differences between the work of an ordinary computer language parser and the morphological parser Morphar. These are described in detail in 4.3.

4.1 The Lexicon

The lexicon has a number of entries, one for each lexeme. The lexeme can be retrieved by its stem. Here, the *stem* is the longest common beginning of all forms (inflected on tense, number, gender etc.) of the lexeme. There may exist zero-length stems.

Every lexeme consists of the *lemma* (the “canonical” form of the word, e.g. the infinitive for verbs or the singular indefinite of the noun) and some inherent properties. One inherent property is the syntactic category. Another is the paradigm.

The paradigm may be shared among all words with the same exact inflected forms, or may be known by a single word only. In the Morphar system, the paradigm consists not only of the inflectional endings (suffixes), but also the *compositional endings*. The compositional endings are all the possible compositional morphemes that are used when the lexeme is the non-last word of a compound. As an example, the word *boll* (ball) has the inflectional ending -s, as in *fot-bolls-skor* (football shoes). It can be noticed that *boll* also have the inflectional ending ∅, as in *boll-plan* (ball park). The use of inflectional endings is not completely arbitrary, but the rules can be quite complex and there is little need to constraint the parser to valid compositional endings only.

4.2 Creating The Lexicon

To make the lexicon memory efficient, we need to keep track of the paradigms created, in order to share the paradigms between words as far as possible. Introducing the syntactic category entity, which is nothing but a list of paradigms, does this. We need one list for each syntactic category.

A paradigm can be constructed if we know all the forms of a lexeme. After computing the stem and all suffixes, we may compare with existing paradigms. If a paradigm matches, we use it. Otherwise, we create a new paradigm with the new suffixes.

The standard paradigms for nouns (declensions) and verb (conjugations) may be added beforehand. In that case, we are able to add more information, for instance gender and compositional endings.

4.3 The Analyser

The analyser used in the Morphar system works almost like a computer language parser. One difference, however, is that each input string may result in several abstract syntax trees. Another difference is that an ordinary parser returns a tree structure for each possible interpretation, but the Morphar analyser returns only a list. So, instead of one syntax tree, we may get several *morpheme lists*.

The analyser computes all the possible stems of a word (that is, all initial substrings of the word) and for each retrieves the lexemes with matching stems in the lexicon. If a lexeme is found, the analyser tries to inflect it using its paradigm to match the input word. If a match is found, it is added to the list of results. Then, if possible, the analyser adds a compositional ending to the stem and concatenates the result so far with the results of the analyse for the rest of the word. In short, the algorithm can be described in the following way:

1. Find all stem candidates.
2. Find every lexeme that has a stem equal to the stem candidates.
3. If a lexeme can be inflected to match the input word, we have found a morpheme list (syntax tree).
4. If a lexeme has a compositional ending that matches the part following immediately after the stem in the input word, repeat recursively from step 2.

When all morpheme lists are found, we should sort them on probability. The user (or client program) may then use a first-N algorithm to consider only the N most probable analyses.

5 Implementation notes

The Morphar reference implementation is programmed entirely in Java. Each model entity (lexicon, lexeme, inflectional ending, paradigm, syntactic category, analyser etc.) is implemented as a class.

The source of the lexicon is the word list used by *Den stora svenska ordlistan*[2]. The word list is distributed under the Creative Commons Share-Alike 1.0 license[3]. The source currently consists of about 25.000 lexemes.

The lexicon is built by a hash table holding lists of lexemes sharing the same stem. The stems are the keys in the table.

The result is returned in a tree structure. The structure can be printed to a PrintStream or a Writer, which can be directed to the console or a text field in a graphical user interface. The abbreviations used in the output is the same as in the Stockholm-Umeå Corpus (SUC) of Written Swedish[4].

The implementation contains two user interfaces. The first is a simple console program, which prompts the user for input and displays the result (the input can be specified on the command line as well). The second implementation is a graphical user interface (GUI) written using Java Foundation Classes (JFC). The GUI can be run stand-alone[5] or as an applet. However, since applets in web browsers are disallowed to access files on disk, the applet version can be run from the AppletViewer tool only . The AppletViewer is included in the Sun Java SDK release.

6 Pros and Cons of The Morphar Model

The Morphar model is simple, yet efficient. The analyser is fast. Words are analysed in milliseconds. The program returns the correct analysis sorted first in 70-80 % of real-world random input words. The reference program loads in under three seconds on any standard performance PC (around 1.000 MHz and 256 MB of internal memory).

Some disadvantages compared to the standard two-level model has shown to exist. In the model, there is no support for derivational morphemes. However, such support can be added without significant changes to the model. Also, the reference implementation returns too many incorrect results. This can be avoided by adding post-analysis rules, as is done in computer language compilers. The rules could prescribe that only a subset of the lexemes and the forms can exist in compounds, and words of syntactic categories A and B cannot be combined to form compound words.

For the reference implementation, it is a shortcoming that the source word list contains only around 25.000 words. Also, information on compositional endings is missing in the source and is added by hand when creating the standard paradigms for nouns and verbs.

As of today, paradigms cannot handle umlauts and ablauts. The 4th conjugation for verbs must thus be treated as many paradigms, one for each lexeme. This, of course, is not a disadvantage of the model, but of the implementation.

7 Conclusion

The Morphar model has proven to be useful and efficient. The reference implementation works satisfactory in most situations, and is pretty fast too. Some features are missing in the model, first and foremost a rule-based filter to remove incorrect analyses. Also, the handling of derivational morphemes is yet to be modelled, and the paradigms need to be refined.

The implementation is working and may be distributed freely for use in non-commercial applications.

Acknowledgement

Thanks to Richard Johansson, Department of Computer Science at The Institute of Technology, University of Lund. Without his supervision the Morphar model would have been less than good.

References

Gerard M Dalgish. Teaching the Computer Swedish: Morphology and Phonology. *CALICO Journal, Volume 7 Number 4.*

Victoria Fromkin and Robert Rodman. 1998. *An Introduction to Language, 6th edition.* Harcourt Brace & Company, Orlando, FL.

Staffan Hellberg. 1978. *The Morphology of Present-Day Swedish.*

Kimmo Koskenniemi. 1997. Representations and Finite-State Components in Natural Language. In Roche, E. and Y. Schabes, editor, *Finite State Natural Language Processing.* MIT Press, pp. 99-116.

[2] Created by Tom Westerberg. See http://sv.speling.org for more information.
[3] The license can be found at http://creativecommons.org/licenses/sa/1.0.
[4] see Appendix A.
[5] see Appendix B.

Pierre Nugues. 2003. *Morphology and Part-of-Speech Tagging*. Draft version.

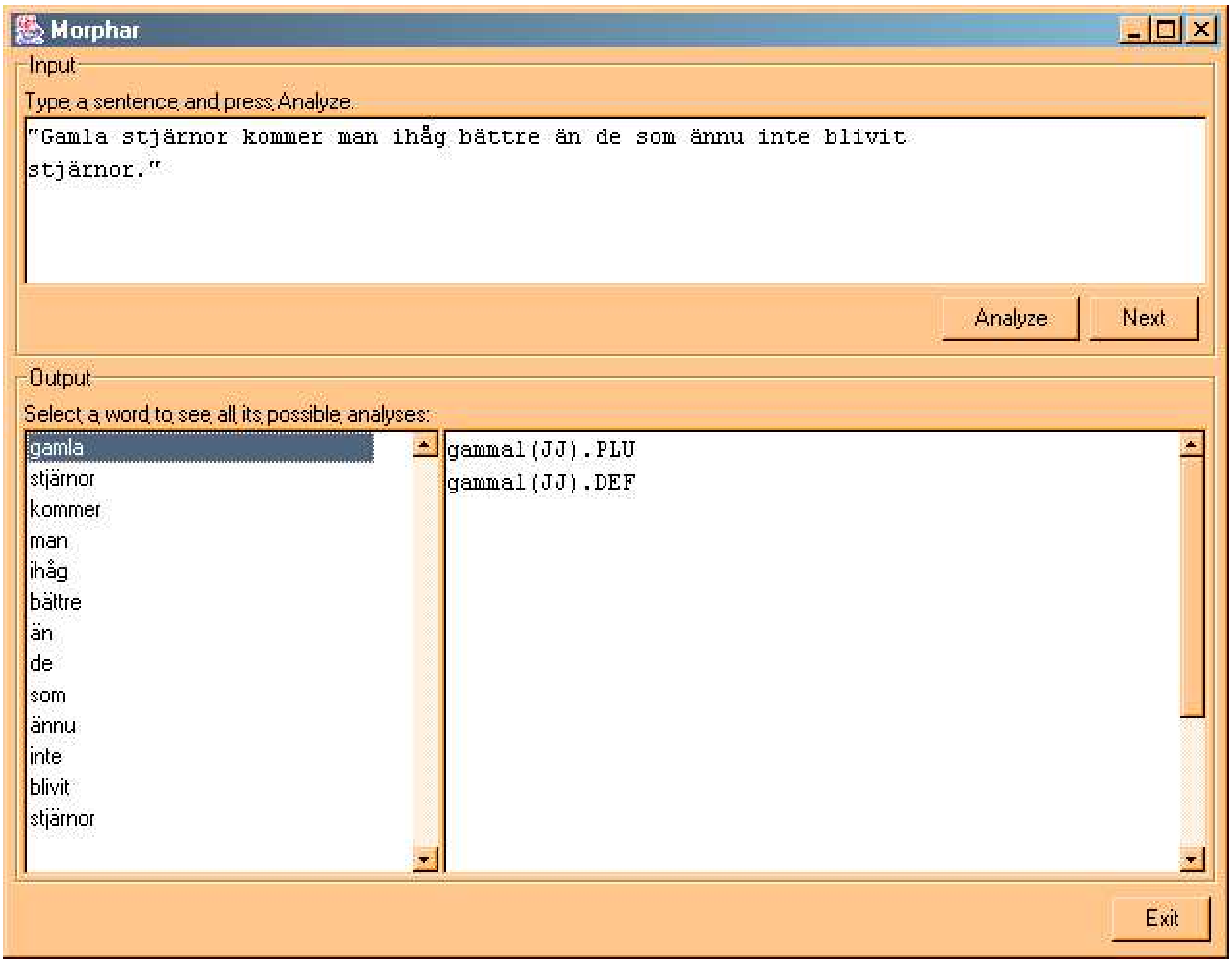
Ulla Stroh-Wollin. 1998. *Koncentrerad nusvensk formlära och syntax*. Studentlitteratur, Lund, SE.

A. SUC Abbreviations

Category Code	Category
AB	Adverb
DL	Delimiter (Punctuation)
DT	Determiner
HA	Interrogative/Relative Adverb
HD	Interrogative/Relative Determiner
HP	Interrogative/Relative Pronoun
HS	Interrogative/Relative Possessive
IE	Infinitive Marker
IN	Interjection
JJ	Adjective
KN	Conjunction
NN	Noun
PC	Participle
PL	Particle
PM	Proper Noun
PN	Pronoun
PP	Preposition
PS	Possessive
RG	Cardinal Number
RO	Ordinal Number
SN	Subjunction
UO	Foreign Word
VB	Verb

Feature Code	Feature	
UTR	Common (Utrum)	Gender
NEU	Neutre	Gender
MAS	Masculine	Gender
SIN	Singular	Number
PLU	Plural	Number
IND	Indefinite	Definiteness
DEF	Definite	Definiteness
NOM	Nominative	Case
GEN	Genitive	Case
POS	Positive	Degree
KOM	Comparative	Degree
SUV	Superlative	Degree
PRS	Present	Verb Form
PRT	Preterite	Verb Form
INF	Infinitive	Verb Form
SUP	Supinum	Verb Form
IMP	Imperative	Verb Form
AKT	Active	Voice
SFO	S-form	Voice
KON	Subjunctive	Mood
PRF	Perfect	Perfect

B. Screen Dump from The Reference Implementation


Morphar
Input
Type a sentence and press Analyze.
Analyze
Next
Output
Select a word to see all its possible analyses:
Exit

```
"Gamla stjärnor kommer man ihåg bättre än de som ännu inte blivit
stjärnor."
```

```
gamla
stjärnor
kommer
man
ihåg
bättre
än
de
som
ännu
inte
blivit
stjärnor
```

```
gammal(JJ).PLU
gammal(JJ).DEF
```

Tagging of named entities in Swedish traffic accident reports

Mats Svensson
Enea Systems AB
Box 4150
203 12 Malmö, Sweden
masv@enea.se

Abstract

A system has been designed to tag named entities (NEs) from text. The relevant domain is traffic accident reports. The texts are written in the Swedish language. The NEs to be tagged are names of roads, streets, city squares, towns and cities.

The system makes use of a rules-based approach. Gazetteers are used to find larger cities, morphological rules are applied to individual words, and context rules are applied to groups of words.

The project has shown evidence that the formation of Swedish words aids in identification and tagging of information in text.

1 Introduction

The task was to develop a system to be incorporated into the CarSim project [1]. For this purpose, a sub-system that can tag towns, cities, roads, highways, streets and city squares was needed.

The task at hand is divided into two parts:

- Detection of roads, highways streets and squares.
- Detection of city and town names.

It turned out that detection of roads, squares and particularly highways can be done with very good results by applying simple regular expressions to text. However, the formation of town names is a lot more complex than the formation of street names. This is due to the fact that many town names are composed of word stems belonging to a much older form of the language in use today.

The working hypothesis at the outset of the project was to use regular expressions to perform the first task, and to use morphological rules combined with context rules to solve the second problem.

2 Related work

This project has been inspired by the work of Nugues et al [1]. Another source of inspiration is the tagging systems developed by Andrei Mikheev and his team described in [4].

3 Regular expressions

In certain parts of this text, regular expressions are used. The syntax used is standard, however a few notations may need a further explanation.

A typical regular expression to match the letters A to Z may be written as [A-Z]. In Swedish, the alphabet is expanded with three more letters after Z, the letters Å Ä and Ö. In some regular expressions in this text, the patterns [A-Ö] and [a-ö] appear, indicating a pattern that would match all characters of the Swedish alphabet.

4 Tags

In order for the CarSim system to be able to operate on the NEs found in the text, these are tagged with XML tags. The XML tag set used was invented for this purpose, but follows the MUC[1] ENAMEX convention. The following tags are used by the system:

`<ENAMEX TYPE="ROAD">`	Country roads.
`<ENAMEX TYPE="HIGHWAY">`	Highways.

[1] Message Understanding Conferences. See [2].

`<ENAMEX TYPE="STREET">`	City streets.
`<ENAMEX TYPE="SQUARE">`	City squares.
`<ENAMEX TYPE="CITY">`	Major city.
`<ENAMEX TYPE="TOWN">`	Towns and minor communities.

The tags are applied to text as follows by this example:

Olyckan skedde på `<ENAMEX TYPE="ROAD">` *väg 16*`</ENAMEX>` *mellan* `<ENAMEX TYPE="TOWN">` *Dalby* `</ENAMEX>` *och* `<ENAMEX TYPE="CITY">` *Lund*`</ENAMEX>`*. Lyckligtvis, för de inblandade, befann sig en polispatrull i närheten på* `<ENAMEX TYPE="STREET">` *Norrängavägen* `</ENAMEX>`*.*

Translation: The accident occurred on road 16 between Dalby and Lund. Fortunately for those involved, a police patrol happened to be nearby on Norrängavägen[2].

5 Order of processing

The system goes through a number of processing steps in sequence to perform the tagging. These steps are:

1) Build up an internal data representation of the text for efficient access to tokens.
2) Perform a pass through the text looking for and tagging county and country roads and highways.
3) Perform a pass through the text looking for and tagging compound street names.
4) Perform a pass through the text looking for and tagging multi-token street names.
5) Use a gazetteer to find and tag the names of major cities.
6) Perform a pass through the text to tag towns based on word prefix and suffix.
7) Perform a pass through the text to find towns by using context rules. In this step, a list is built of possible towns.
8) Pass through the list of possible towns and clean out all duplicates and known false hits.
9) Use list of towns as a gazetteer to tag the text for towns in a final pass.
10) Print out tagged representation of text.

At each tagging stage, previously tagged groups of tokens are ignored by the system in order to speed up processing.

6 Finding roads and highways

Swedish county and country roads as well as highways follow very simple rules of naming. This makes detecting them quite trivial.

The common way for country and county roads to be mentioned is as *väg 16*, *länsväg 16* or *riksväg 108* with the possible abbreviations *lv 16* and *rv 108* for the last two[3].

These can be found by looking for the following patterns in the text:

```
väg Nn
väg Nnn
riksväg/rv Nn
riksväg/rv Nnn
länsväg/lv Nn
länsväg/lv Nnn
```

Here, **N** denotes a digit from 1 to 9 and **n** denotes a digit from 0 to 9. The slash denotes a choice between any of the words it separates.

Finding Swedish highways is similar to the task of finding roads. Swedish highways are named *E6*, *E22* and the like, an *E* followed by one or two digits, where the first one is always non-zero.

The highway tagger looks for the following patterns:

```
EN                    E Nn
Enn                   EN:an
motorväg EN           ENn:an
mororväg ENn          E N:an
Väg EN                E Nn:an
Väg ENn
E N
```

[2] *Dalby* is a minor town, *Lund* is a city and *Norrängavägen* is the name of a street.

[3] *väg 16* means *road 16, länsväg 16* means *county road 16* and *riksväg 108* means *country road 108*. The abbreviations are more common in police reports and are never used in newspaper text.

These patterns have proven to be adequate for tagging all roads and highways in text taken from the relevant domain. They achieved a 100% precision and 100% recall score in a test where a 290 kB development corpus of relevant domain-specific text was tagged[4].

7 Finding compound street names

The Swedish language very often uses compounding to form new words, quite the opposite to what is done in Romance languages and English.

An example is a person who works with cleaning windows. He is referred to as a *window cleaner* in English, but in Swedish, the word *fönsterputsare* is formed by the words *fönster* (window) and *putsa* (to clean or to polish something).

This compounding property of the language also reflects how city streets and town squares are named. A typical Swedish street may be named after an entity compounded with a word for a street.

For example a street called *Oak street*, named after the oak tree would be called something like *Ekvägen* in Swedish, where *Ek* is Swedish for the oak tree or oak wood, and *vägen* is the nominative form of a word meaning *street*.

The compound street tagger uses this regularity of the language to find roads and city squares. The following regular expressions are used to search for target expressions:

[A-Ö][a-ö]*gatan	[A-Ö][a-ö]*vägen
[A-Ö][a-ö]*stigen	[A-Ö][a-ö]*torget
[A-Ö][a-ö]*platsen	[A-Ö][a-ö]*gränd
[A-Ö][a-ö]*gränden	[A-Ö][a-ö]*leden

These regular expressions may of course score hits that prove to be wrong. A good example is that we want to be able to detect and tag *Götaplatsen*, a square in the city of Göteborg. However, within the domain, a very common word is *olycksplatsen* (the scene of the accident). When this starts a sentence, the word is capitalized, which will cause a false hit.

In order to get around this, the system uses a gazetteer of false hits, an exclusion list. Tokens or groups of tokens are compared against exclusion lists to avoid tagging words such as *Olycksplatsen*.

Of course, each such exclusion list is both specific to the domain and specific to the tagging task at hand. The contents of such a list along with the design of the regular expressions that it guards can be seen as the training of the system.

8 Other street names

Compounded words are not the only way to form street names in Swedish. A very small fraction of street names are formed completely irregularly, but most still contain words like *väg*, *gata* (road, street), *stig* (path) and so on.

An extension to the notion of compound street names is multi-token street and square names, henceforth referred to as just multi-token names.

A multi-token name is a street such as *Ernst Wigforss gata* (a street in Lund), *Lilla Torg* (a square in Malmö) and so on, composed of several tokens.

These are easy to detect by using regular expressions and exclusion lists. These are some of the rules used to find such streets:

[A-Ö][a-ö]* [A-Ö][a-ö]*sgata	[A-Ö][a-ö]*s gata
[A-Ö][a-ö]* [A-Ö][a-ö]*sgatan	[A-Ö][a-ö]*s gränd
[A-Ö][a-ö]* [A-Ö][a-ö]*sgränd	
[A-Ö][a-ö]* [A-Ö][a-ö]*sväg	
[A-Ö][a-ö]* [A-Ö][a-ö]*s väg	
[A-Ö][a-ö]* [A-Ö][a-ö]*s gata	

Examples of exclusion list entries for these rules are expressions such as *På väg* (on the way), *Hans väg* (His path) and similar, common constructs of the language that deal with the word *väg* (road, path).

Similar rules work fine for squares, again keeping in mind that *plats* (a synonym for square in Swedish) is also a common word meaning *place* or

[4] The test corpus consisted of press clippings on car accidents from Swedish news sources.

location, which has to be kept in mind when designing the exclusion list for this rule set.

9 Tagging major cities

The system uses a gazetteer to find those tokens that are to be tagged as CITY. The reason for this is that there are not many major cities in Sweden. The test system developed used a gazetteer of 22 cities for this purpose.

The system will look for two things when using the gazetteer. Not only is the exact match to the listed item sought, but also the word with the letter 's' appended to form the possessive form of the word.

The system will find the city *Stockholm* if an exact match can be done or a match can be done to the variant *Stockholms*.

Swedish is abundant with compounded words, as has already been noted. A phrase such as *the police in Stockholm* is built up as one word, *Stockholmspolisen* in Swedish. The system will not, and shall not tag such compounds due to the fact that they seldom refer directly to a location but rather to where a certain subject is from.

10 Tagging towns using prefixes and suffixes

As in many other European languages, Swedish town names often stem from words that were in common usage hundreds of years ago, creating patterns in their formation. This, combined with the Swedish quirk for compounding words, is very useful when it comes to searching for town names.

Although there are local themes and variations on formation of town names, most notably far north, where naming often stems from the Saami language rather than from Swedish, morphology can be used to detect town names. Most notable in Swedish town name formation is the plethora of common prefixes and suffixes.

The system works with two lists of suffixes and prefixes, which are matched against capitalized words in the text. This is combined with exclusion lists containing common words that may be formed along the same rules.

Some examples of common suffixes used are:

-arp	-by
-stad	-fors
-boda	-vik
-sala	-järvi

These are present in town and city names such as *Genarp*, *Hyby*, *Filipstad*, *Bengtsfors*, *Lönsboda*, *Valdemarsvik*, *Onsala* and *Jukkasjärvi*.

When it comes to prefixes, very many of the more common prefixes involve names of kings or other historical persons as exemplified by the cities *Karlstad* and *Karlskrona*.

However, it turned out that the suffix proved to be a better source for identification of a town name than the prefix. The final system looks only for these prefixes:

Mal-	Kung-
Kristian-	Karl-
Chistian-	Carl-
Näs-	Berg-
Kal-	

It is quite possible to find a lot more common prefixes on Swedish towns, however, extending the prefix list when using a large suffix list added very little to the detection scores of the program.

One large cause for false hits by using the suffix rules was that very many surnames of people are formed in the same way as town names. *Lindesberg* is a town, whereas *Magdalena Forsberg* is a person. A further complication is that the target domain consists of traffic accident reports from newspapers and similar sources, where people often are introduced once with full name, and thereafter mentioned only by last name.

To avoid scoring false hits due to this kind of complications, the system tries to determine whether a name, rather than a town, has been found. If a possible town is preceeded by a capitalized word, this word is checked against a list of common first names. If there is a match, the word is considered to be a name rather than a town.

Another method used to find names is based on the Swedish tradition of double names. Double names in Swedish are first names that are hyphenated, like *Jan-Åke* and *Anna-Karin*. If a possible town is preceeded by two capitalized words, which are separated by a hyphen, it is also considered to be a name.

If a word has been categorized as being a name and not a town, a backwards search is done about 50 words, and a forward search about 250 words to remove false tagging of the name. The reason for this is that it is very common for newspaper text to omit the first name of a person in the headline, introduce the person by full name somewhere near the beginning of the text and then refer to the person by last name only for about one quarter of a page until the full name is mentioned again.

11 Tagging towns using context rules

Very many Swedish towns can be found by the simple rules described above. However, there are other town names that are not as regularly formed. Whereas *Bjärred* can be found by the suffix rules and *Kungsör* can be detected by the prefix rules, the town of *Lomma*, which is close to Bjärred geographically, will not be detected. None of the towns *Morgongåva*, *Virke* or *Råå* would be found by those rules either.

In order to improve detection, a set of context rules was introduced. These are used to analyze the surroundings of a word. In the domain of traffic accident reports, town names are commonly used in certain special contexts.

The context rules are applied by scanning the text for matching patterns. As soon as a complete match is made, the words that may possibly be towns are extracted. These are then added to a list of town candidates for further processing.

These are examples of context rules[5]:

polisen i **TOWN1**
ROAD1 mellan **TOWN1** och **TOWN2**
norr om **TOWN1**
trafikolycka i **TOWN1**
bilolycka i **TOWN1**
STREET1 i norra **TOWN1**

Note the type tags: ROAD1 will match any token or group of tokens tagged as a road by the earlier passes that the system made over the text. Also, STREET1 will match anything tagged as a street.

After the list of town candidates has been built, it is subject to elimination of duplicates. Any town found should only be on the list once.

After this, all already known cities are eliminated, as are all towns that match known prefixes and suffixes. Exclusion lists are also applied, both on suffixes and on entire words, to remove references to gas stations, forests, streets, sports arenas and so on. The reason that the exclusion lists operate on suffixes is the affinity for compounding words in Swedish.

As an example, we want to find the town *Virke* in this text: "*Olyckan inträffade på väg 104 mellan Virke och Stora Harrie. Mer exakt inträffade olyckan strax norr om Shellmacken.*"[6]. We do not, however, want to get the word *Shellmacken* on the towns list[7].

The final result of the application of context rules and post-processing is a list of towns. This list is used to make a pass through the text and tag all occurrences of those words.

[5] The translations are The police in TOWN1, ROAD1 between TOWN1 and TOWN2, North of TOWN1, Traffic accident in TOWN1, Car accident in TOWN1 and STREET1 in northern TOWN1.
[6] *The accident occurred on road 104 between Virke and Stora Harrie. More specifically, it happened west of the Shell gas station*. This sentence is, however, constructed. There are no gas stations in Virke, and road 104 does not run through the town.
[7] The word means *the Shell gas station,* and is another example of compounding of words in the Swedish language.

A test on the 290 kB development corpus showed that 50 towns that could not otherwise be detected were found by this set of rules, of which 11 were false hits. All false hits were actual geographic locations, however. Some were lakes and islands, some were foreign countries and some were city boroughs[8].

12 Multi-token towns

Some towns in Swedish are augmented with extra, descriptive words. Examples are *Lilla Edet*, *Stora Råby* and *Södra Sandby*. Common for many of these is that the first word is almost always an adjective like *Norra* (northern) or *Lilla* (lesser).

Therefore, the system is instructed to look for a list of adjectives before any towns that are found. If a capitalized adjective that is on the list is found before the town name, this is also incorporated in the tag.

13 Results

The final system was blind-tested on texts it had not been subject to before. A 14 kB text was used, which was composed of domain-specific text from Swedish news sources. The total number of tokens in the text (words and punctuation) was 2533.

On this text, precision and recall was measured. Recall is measured as the number of relevant results in the answer set over the total number of possibly relevant results. Precision is measured as the ratio of relevant results over all results in the answer set.

For the test text, a recall of 93% and precision of 89% was measured. Upon inspection, the text showed evidence that the results could be improved by adding more context rules and restricting more words by using the exclusion lists for town suffixes[9].

14 Conclusions

The project has shown that detection and tagging of Swedish towns, streets, roads and cities can be done quite reliably by using a rules-based approach. Evidence was found that roads and highways can be automatically detected with absolute confidence by applying simple rules based on regular expressions. City squares and streets may also be detected quite reliably by applying slightly more advanced regular expressions.

When it comes to detecting and tagging towns, the approach that worked best was to look for common suffixes. Prefixes added a bit more recall (it went up from 89% to 93%), and precision was helped by the addition of the exclusion lists (an improvement from 81% to 89%).

The towns that were not detectable by use of prefix and suffix rules could in some cases be found by the context rules instead. It turned out that the few simple context rules used for the test system worked well enough to tag only geographical locations, however, only about 80% of the locations they managed to detect were actually towns.

References

[1] Per Andersson, 2003, *A Prototype to Extract and Visualize Information from Car Accident Reports in Swedish*. Master's Thesis, department of Computer Science, Lund Institute of Technology.

[2]**WEB:**

http://www.cs.nyu.edu/cs/faculty/grishman/muc6.html

[3] Jerry R. Hobbs, Douglas Appelt, John Bear, David Israel et al., 1996, *FASTUS: A Cascaded Finite-State Transducer for Extracting Information from Natural Language text* in Roche and Schabes, eds., *Finite State Devices for Natural Language Processing*, MIT Press, Cambridge MA, 1996

[4] Andrei Mikheev, Marc Moens and Claire Grover, 1999, *Named Entity recognition without gazetteers* in *EACL'99, Bergen, Norway* ACL June 1999. pp. 1-8

[8] It may, with good reason, be possible to argue that a city borough is a town.

[9] It shall be kept in mind that the system, as described here, is a small prototype system.

A Writing Assistent Using Language Models Derived From the Web

Sharon Tsai
d00yt@efd.lth.se

Abstract

In the field of Linguistic there exists many powerful tools for measuring the statistic characteristics of words and sentences. These tools rely on a corpus to which the data is compared. In order to get good and meaningful results from the tools available, a suitable corpus is thus needed. As the corpus is the key that ties the tools together, it is of uttermost importance. For most applications, all though not all, a large corpus is useful. This paper presents a solution to using the largest corpus known to man, the Internet. It will show a prototype program using many different linguistic tools on information gathered by the premiere search engine Google.

1 Method

The basic mathematical tools needed for this work is explained in this section. For a more in depth explanation please see (Language Processing Computational Linguistics, 2003).

1.1 N–Grams

N–Grams is simply a method of counting the frequency of a sequence of N word in the corpus. These frequencies can then be used to calculate probabilities. Equation 1 shows the probability of a single word occurring. It is naturally the frequency of the word occurring in the corpus divided by the amount of different words available in the corpus.

$$P(w) = \frac{C(w)}{N} \tag{1}$$

It is also possible to calculate probabilities of a word following a given word. This is known as the maximum likelihood estimate and is defined in equation 2.

$$P(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})} \tag{2}$$

The maximum likelihood estimate for a word following two given words is defined in equation 3.

$$P(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})} \tag{3}$$

1.2 Mutual Information

Mutual Information is a tool for measuring the strength of word associations. A high value is an indication of two words occurring together but with a total small frequency, such as technical terms. For instance, such terms might be ”hyperthreading processor” or ”keyhole surgery”. Mutual Information is defined in equation 4

$$I(w_i, w_{i+1}) = \log_2 \frac{NC(w_i, w_{i+1})}{C(w_i)C(w_{i+1})} \tag{4}$$

1.3 T–Score

T–Score is a statistic tool which measures frequently occurring grammatical combinations. A high T–Score means that the two words occur often together, such as ”of the” and ”in the”. The definition of T–Score is shown by equation 5

$$T(w_i, w_{i+1}) = \frac{C(w_i, w_{i+1}) - \frac{1}{N}C(w_i)C(w_{i+1})}{\sqrt{C(w_i, w_{i+1})}} \qquad (5)$$

2 Implementation

2.1 Module Overview

The prototype program is implemented in Java. For more information please see (J2SE 1.4.1 API Specification, 2003). It is made up by five main classes: GUI, UserPane, SearchResultPane, SearchHandler and HTMLWriter. See figure 1 for the overview of the structure.

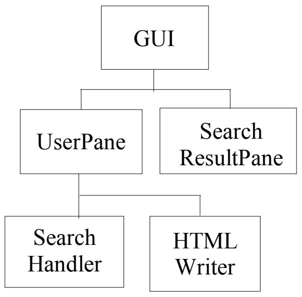


Figure 1: An overview of the architecture

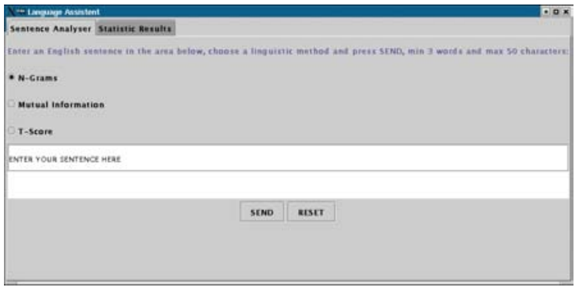
2.2 User Interface

2.2.1 GUI

The GUI is a frame that contains a UserPane and a SearchResultPane. A user can change view by clicking on the respective tab that represent UserPane and SearchResultPane.

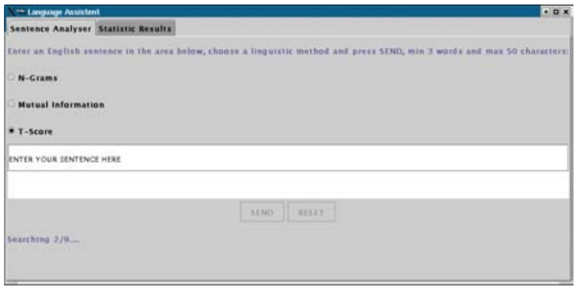
2.2.2 UserPane

The UserPane consists of a button group, a text field, a text area, two buttons, Send and Reset, and a status label. The button group contains a list of linguistic methods that a user can choose to analyze the input sentence with. Only one method can be chosen at the same time. These methods that are included in the prototype program are N-Gram, Mutual Information and T-Score. The text field is where the user types in the sentence which is to be analyzed. The final result is presented in the text area below the text field. The status label at the bottom updates the program status, search progress and possible error messages. See figure 2, 3 and 4.



Enter an English sentence in the area below, choose a linguistic method and press SEND, min 3 words and max 50 characters:

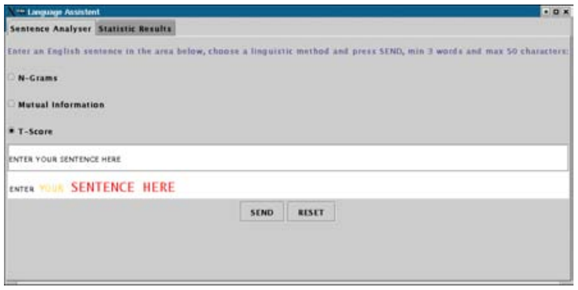
Figure 2: GUI: UserPane



Enter an English sentence in the area below, choose a linguistic method and press SEND, min 3 words and max 50 characters:

Searching 2/9....

Figure 3: GUI: UserPane during search



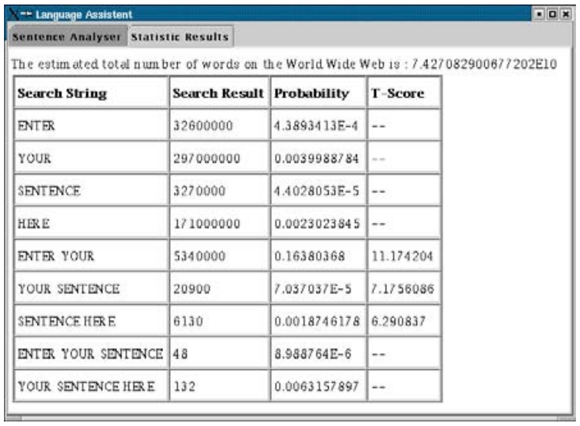
Enter an English sentence in the area below, choose a linguistic method and press SEND, min 3 words and max 50 characters:

ENTER YOUR SENTENCE HERE

Figure 4: GUI: UserPane after executing search

2.2.3 SearchResultPane

The SearchResultPane consists of a web page that is generated automatically in the end of every search. See figure 5. The web page reloads automatically after every new search and contains all the statistical data gathered.



The estimated total number of words on the World Wide Web is : 7.427082900677202E10

Search String	Search Result	Probability	T-Score
ENTER	32600000	4.3893413E-4	--
YOUR	297000000	0.0039988784	--
SENTENCE	3270000	4.4028053E-5	--
HERE	171000000	0.0023023845	--
ENTER YOUR	5340000	0.16380368	11.174204
YOUR SENTENCE	20900	7.037037E-5	7.1756086
SENTENCE HERE	6130	0.0018746178	6.290837
ENTER YOUR SENTENCE	48	8.988764E-6	--
YOUR SENTENCE HERE	132	0.0063157897	--

Figure 5: GUI: SearchResultPane

2.3 Implementation Detail

The data flow chart, figure 6, shows all seven stages of the prototype program. See the corresponding sections for further detail and information on each stage.

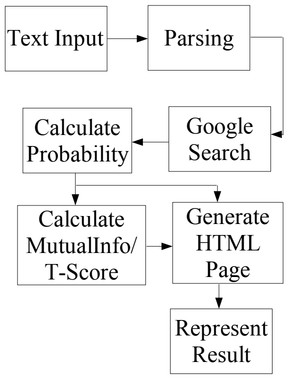


Figure 6: Flow chart

2.3.1 Initiating GUI

During the program initiation, the total number of words that exists on the Internet is estimated. This is done by calculating the occurring percentage of words such as ”in,” ”on,” and ”of” in a fixed size English corpus. This percentage is then applied on the sum of the Google search results of these words in order to estimate total number of words. The percentage used in this program is obtained through Jane Austin’s novel ”Emma.” There are around 74.6 billion words on the Internet and the number is growing everyday.

2.3.2 Text Input

This stage is the very beginning of the program cycle. It is done simply by clicking on the Send button or press ”ENTER” on keyboard after entering a sentence in the text field. At the moment, the size of the sentence is limited at 50 tokens. The sentence is then sent to the SearchHandler object.

2.3.3 Parsing

The SearchHandler first applies an error control on the incoming sentence. The main purpose of the error control procedure is to filter out the word ”the,” which causes overflow in the automated Google search. The SearchHandler also checks if the sentence is empty. If any of these two errors occurs, the program is terminated and an appropriate error message displays at the status label. The sentence is then parsed into unigrams, bigrams and trigrams if no errors are found. All N-Grams are stored in an array.

2.3.4 Google Search

The SearchHandler traverses the N-Gram array and performs Google search on every N-Gram. Google returns an estimated result back and it is saved in the corresponding position in a separate array for search results.

Google’s own API is used in this program because Google does not permit automated queries without an API account. Communication is performed via Simple Object Access Protocol(SOAP). For further information about Google API see (Google API, 2003).

2.3.5 Calculate Probability

No matter which linguistic method is chosen to analyze the input sentence, N-Gram probabilities are always calculated according to formulas presented in section 1. The resulting N-Gram probabilities are stored in a separate array in the corresponding position.

2.3.6 Calculate Mutual Information/ T-Score

If the chosen linguistic method to analyze the input sentence is Mutual Information or T-Score, values for bigrams are calculated according to formulas presented in section 1. These values are also stored in a separate array.

2.3.7 Generate HTML Page

After all the necessary values are calculated, a web page is generated by HTMLWriter object and loaded into the SearchResultPane. It can also be viewed by any web browser. The web page contains the estimated total number of words on the Internet, and a table of all N-Grams and their corresponding N-Gram probability and Information/ T-Score values, if any.

2.3.8 Represent Result

Depending on which linguistic method is chosen, the corresponding result array is traversed. Separate strategies are used to analyze results of N-Gram, Mutual Information and T-Score. There are two ways to represent unusual words, in orange style and red style. The font of words is slightly bigger than default, size sixteen, and painted in orange. In red style, it is size twenty font and painted in red.

For N-Gram, if any of the N-Gram probability is less than 0.1%, it is marked in orange style. If the N-Gram probability is less than 0.05%, it is marked in red style.

The bigram that has the highest Mutual Information value is marked in red style and the second highest bigram is marked in orange style.

The marking strategy for T-Score is the opposite for Mutual Information. The bigram that has the lowest T-Score value is marked in red style and the second lowest bigram is marked in orange style.

3 Experiments

3.1 Experiment I: ”I really like strawberry beer”

3.1.1 Idea

The sentence, ”I really like strawberry beer,” is used as reference sentence during the development phase because the very rare occurrance/use of the word combination ”strawberry” and ”beer.”

3.1.2 Search Results

As figure 7 clearly shows that all three methods manage to detect the unusual word combination, ”strawberry beer.” This is the primary reason that the Mutual Information marking strategy is chosen to mark the highest and the second highest value.

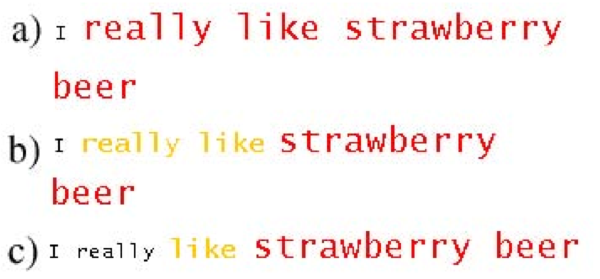

a) I really like strawberry beer
b) I really like strawberry beer
c) I really like strawberry beer

Figure 7: Search results: The sentence is analyzed by: a) N-Gram b) Mutual Information c) T-Score

3.1.3 Conclusion

In this example, the program demonstrates the ability to detect strange or unusual word combinations.

3.2 Experiment II: ”This project consists of three parts, and it consists at working”

3.2.1 Idea

For most people, the most difficult part of learning a foreign language is prepositions. In this example, the most common prepositions that occur with the word ”consist” are ”of” and ”in,” according to Oxford Advanced Learner’s Dictionary. The goal is that the program should be able to spot and mark the bigram, ”consists at” in red style, preferably, or in orange style because it is grammatically incorrect.

3.2.2 Search Results

Figure 8 shows that linguistic methods N-Gram and T-Score manages to marks the error bigram in red style, however, Mutual Information fails to produce correct results.

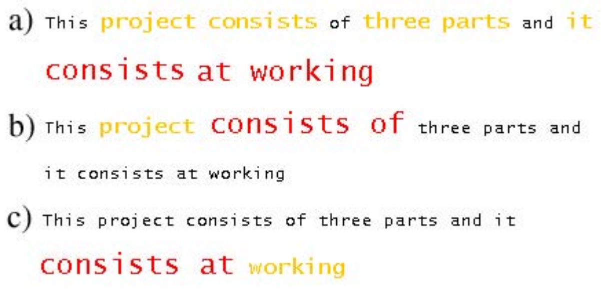
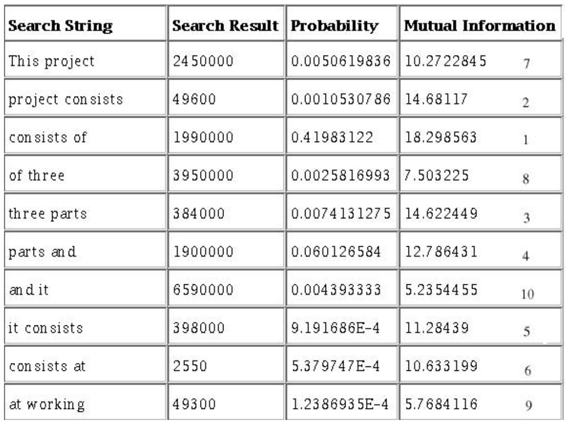

a) This project consists of three parts and it consists at working
b) This project consists of three parts and it consists at working
c) This project consists of three parts and it consists at working

Figure 8: Search results: The sentence is analyzed by: a) N-Gram b) Mutual Information c) T-Score

With a closer look on the numerical values of the search result which presents in figure 9, the Mutual Information value of the error bigram is ranked the sixth highest or the fifth lowest of all ten bigrams which places it in the middle. Therefore, the error cannot be spotted by using Mutual Information method with the current strategy, whether by marking the bigram that has the highest or the lowest Mutual Information value.



Search String	Search Result	Probability	Mutual Information	
This project	2450000	0.0050619836	10.2722845	7
project consists	49600	0.0010530786	14.68117	2
consists of	1990000	0.41983122	18.298563	1
of three	3950000	0.0025816993	7.503225	8
three parts	384000	0.0074131275	14.622449	3
parts and	1900000	0.060126584	12.786431	4
and it	6590000	0.004393333	5.2354455	10
it consists	398000	9.191686E-4	11.28439	5
consists at	2550	5.379747E-4	10.633199	6
at working	49300	1.2386935E-4	5.7684116	9

Figure 9: Mutual Information values in SearchResultPane with additional rankings

3.2.3 Conclusion

This example demonstrates the program's ability to spot possible incorrect prepositions in a sentence. However, T-Score and N-Gram methods produce better and more reliable results than Mutual Information.

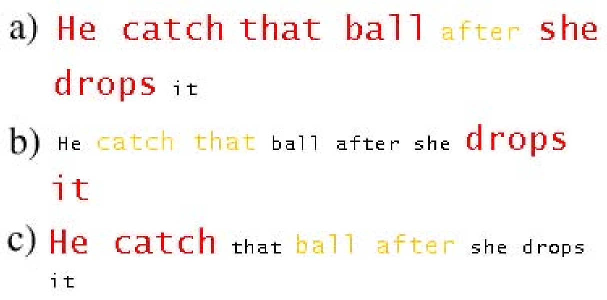
3.3 Experiment III: ”He catch that ball after she drops it”

3.3.1 Idea

There is an obvious tense error in the sentence. It should be ”He catches” instead of ”He catch.” With the help of any of these three linguistic methods, hopefully the program is able to detect this grammatic error.

3.3.2 Search Results

As figure 10 shows that all three methods manages to spot and mark the error bigram in some way. Again, T-Score produces the best results and is still the most powerful linguistic method compared to N-Gram and Mutual Information. Mutual Information only marks ”catch” in orange style, not in red. It also misleadingly marks the bigram ”drops it” in red style, which is grammatically correct.



a) He catch that ball after she drops it

b) He catch that ball after she drops it

c) He catch that ball after she drops it

Figure 10: Search results: The sentence is analyzed by: a) N-Gram b) Mutual Information c) T-Score

3.3.3 Conclusion

The program also has the ability to analyze and check the verb tense in a sentence. However, it can only analyze a sentence at lexical and grammatic level, not in semantic level as it does not try to understand the meaning of the sentence.

4 Further work

There are a few further improvements that can be done in the program. Breaking up the option N-Gram into three different sub-options, unigram, bigram and trigram and during the parsing stage, the input sentence will only be parsed according to the linguistic methods, e.g. only bigrams in T-Score and bigram options and trigrams when the trigram option is selected. This might decrease the run time a little bit, but not very significantly.

If a user wishes to analyze the same sentence with different methods in different search, the program should be able to use the already existed search results from the previous search and calculate new values according to the selected method. This will decreases the run time significantly, especially after the first initial search.

More sophisticated marking strategies can be developed for all the methods instead of the existing straight-forward approach as in T-Score, the lowest in red style, and in N-Gram, under 0.05% marked in red style.

5 Conclusion

This paper has clearly showed that it is possible to implement a language assistance using linguistic methods without a fixed size corpus. The Internet is without doubt the biggest corpus ever created. The number of web pages on the Internet increases everyday. However, the difference in quality and the form and use of the language of these home pages are substantially large. Without carefully analyzing and filtering out the unwanted information, the results may be misleading. The search results can very well represent the most modern, normal-everyday-life, down-to-earth form of a language.

It has been a great learning experience and challenge in both applying textbook formulas into real-life uses and designing a user-friendly program. A few mistakes has been made and corrected, but the most important of all, is the satisfaction of enjoying the final fruit.

References

Pierre Nugues. 2003. *Compendium, Language Processing Computational Linguistics*

Sun Microsystems. 2003. *J2SE 1.4.1 API Specification* java.sun.com

Per Holm 2000 *Objektorienterad programmering och Java*

Oxford University Press 2003 *Oxford Advanced Learner's Dictionary* www.oup.com/elt/global/products/oald/

Google 2003 *Google API* www.google.com/apis